ПОЧУВСТВУЙ КЛАСС

Bertrand Meyer

TOUCH OF CLASS

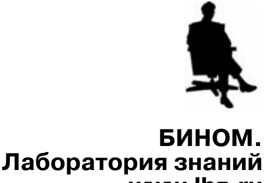
Learning to Programm Well with Objects and Contracts

Бертран Мейер

ПОЧУВСТВУЙ КЛАСС

Учимся программировать хорошо с объектами и контрактами

Перевод с английского под редакцией к.т.н. В.А. Биллига



Национальный Открытый Университет «ИНТУИТ»
www.intuit.ru

БИНОМ. Лаборатория знаний
www.lbz.ru

Москва
2011

УДК 004.42(07)
ББК 32.973.26-018я7
М45

Мейер Б.
М45 Почувствуй класс / Мейер Б.; пер. с англ. под ред. В.А. Биллига.—М.: Национальный Открытый Университет «ИНТУИТ»: БИНОМ. Лаборатория знаний, 2011. —775 с.: ил., табл.

ISBN 978-5-9963-0573-5

В книге обобщен многолетний опыт обучения программированию в ETH, Цюрих. В ней удачно сочетаются три грани, характерные для профессионального программирования, – наука, искусство и инженерия. Она в первую очередь ориентирована на студентов, обучающихся в области информационных технологий, и их преподавателей, но представляет несомненный интерес для всех программистов, создающих программный продукт высокого качества.

В книге излагаются основы объектно-ориентированного программирования (ООП). Особое внимание уделяется корректности программ за счет введения контрактов – предусловий, постусловий методов класса, инвариантов классов. Глубоко и подробно рассматриваются такие механизмы ООП, как наследование и универсальность. Изучаются алгоритмы и структуры данных – массивы, кортежи, списки, хэш-таблицы, различные виды распределителей, деревья. Подробно рассматриваются рекурсивные алгоритмы и рекурсивные структуры данных. Даются основы лямбда-исчисления и вводятся агенты, поддерживающие функциональный тип данных.

Язык Eiffel используется как рабочий язык программирования.

Книга содержит предисловие и шесть частей. Шестая часть содержит пять приложений, в которых дается сравнительный анализ языков программирования – Java, C#, C++, C.

УДК 004.42(07)
ББК 32.973.26-018я7

Полное или частичное воспроизведение или размножение каким-либо способом, в том числе и публикация в Сети, настоящего издания допускается только с письменного разрешения Национального Открытого Университета «ИНТУИТ».

По вопросам приобретения обращаться:
«БИНОМ. Лаборатория знаний»
Телефон (499) 157-1902, (499) 157-5272,
e-mail: binom@Lbz.ru, http://www.Lbz.ru

ISBN 978-5-9963-0573-5

© Springer Verlag, 2009
© Мейер Б., 2011
© Русский перевод.
Национальный Открытый
Университет «ИНТУИТ», 2011

Об авторе

Бертран Мейер, став преемником Николаса Вирта, с 2003 года возглавляет знаменитую кафедру Software Engineering в ETH, Цюрих, – в одном из старейших университетов Европы. Он автор языка Eiffel, основатель и научный руководитель фирмы Eiffel Software (www.eiffel.com).

Бертран Мейер внес неоценимый вклад в развитие теории, практики и технологии объектно-ориентированного программирования. За свою книгу Object-Oriented Software Construction, которая по праву считается библией ОО-программирования, он получил премию JOLT. За заслуги в этой области стал первым лауреатом престижной премии Дала-Нигарда (авторов первого ОО языка – Симулы).

Технология «проектирования по контракту» (Design by Contract), предложенная Бертраном Мейером и в полной мере реализованная в языке Eiffel и в среде разработки EiffelStudio, оказала серьезное влияние на повышение общего качества программных продуктов в современной индустрии ПО.

Бертран Мейер – автор многочисленных научных работ, серии книг (три его книги переведены на русский язык), организатор конференций, редактор журнала по ООП, лауреат премий ACM, Mills и других.

Давняя дружба связывает его с Российским программистским сообществом. В молодые годы он проходил стажировку в Академгородке у А.П. Ершова, является почетным доктором Санкт-Петербургского государственного университета ИТМО.

В предлагаемой читателю книге обобщен опыт восьмилетнего обучения программированию в ETH. Как сформулировал автор, цель этой книги – не просто дать основы инженерии программ и получить опыт создания работающих программ, но показать красоту принципов, методов, алгоритмов, структур данных и других определяющих дисциплину инструментов. И, может быть, самая главная цель – привить чувство, заставляющее вас делать не просто хорошее ПО, а выдающееся, и стремление создавать программы высочайшего качества.

Оглавление

Доступные ресурсы 13

Предисловие редактора перевода 15

Так учат в ETH 15

Предисловие автора к русскому изданию 18

Предисловия 20

Предисловие для студентов 21

Программное обеспечение (ПО, software) повсюду 21
Любительская и профессиональная разработка ПО 22
Предшествующий опыт 22
Современная технология ПО 23
Объектно-ориентированное конструирование ПО 24
Формальные методы 24
Учимся, создавая 24
От потребителя к поставщику 25
Абстракция 26
Цель: качество 26

Предисловие для преподавателей 28

Вызовы первого курса 28
Извне – Внутрь: Обращенный учебный план 31
Выборы технологий 34
Насколько формально? 40
Другие подходы 41
Покрытие тем 42

Благодарности 44

Литература 47

Заметки для преподавателей: что читать студентам? 48

Часть I **Основы** **51**

1 Индустрия чистых идей 51

1.1. Их машины и наши 51
1.2. Общие установки 53
1.3. Ключевые концепции, изученные в этой главе 59

2 Работа с объектами 62

2.1. Текст класса 62
2.2. Объекты и вызовы 64
2.3. Что такое объект? 70
2.4. Методы с аргументами 74
2.5. Ключевые концепции, изученные в этой главе 76

3 Основы структуры программ 79

3.1. Операторы (instructions) и выражения 79
3.2. Синтаксис и семантика 80
3.3. Языки программирования, естественные языки 81
3.4. Грамматика, категории, образцы 82
3.5. Вложенность и структура синтаксиса 83
3.6. Абстрактные синтаксические деревья 84
3.7. Лексемы и лексическая структура 86
3.8. Ключевые концепции, изученные в этой главе 88

4 Интерфейс класса 90

4.1. Интерфейсы 90
4.2. Классы 92
4.3. Использование класса 93
4.4. Запросы 97
4.5. Команды 101
4.6. Контракты 102
4.7. Ключевые концепции, изученные в этой главе 108

5 Немного логики 111

5.1. Булевские операции 112
5.2. Импликация 120
5.3. Полустрогие булевские операции 124
5.4. Исчисление предикатов 128
5.5. Дальнейшее чтение 133
5.6. Ключевые концепции, изучаемые в этой главе 133

6 Создание объектов и выполняемых систем 138

6.1. Общие установки 138
6.2. Сущности и объекты 140
6.3. VOID-ссылки 141
6.4. Создание простых объектов 147
6.5. Процедуры создания 151
6.6. Корректность оператора создания 155

6.7. Управление памятью и сборка мусора 157
6.8. Выполнение системы 158
6.9. Приложение: Избавление от void-вызовов 164
6.10. Ключевые концепции, изучаемые в этой главе 165

7 Структуры управления 167

7.1 Структуры для решения задач 167
7.2. Понятие алгоритма 168
7.3. Основы структур управления 172
7.4. Последовательность (составной оператор) 174
7.5. Циклы 178
7.6. Условные операторы 196
7.7. Низкий уровень: операторы перехода 203
7.8. Исключение GOTO и структурное программирование 206
7.9. Вариации базисных структур управления 211
7.10. Введение в обработку исключений 218
7.11. Приложение: Пример удаления GOTO 223
7.12. Дальнейшее чтение 225
7.13. Ключевые концепции, изучаемые в этой главе 226

8 Подпрограммы, функциональная абстракция, скрытие информации 229

8.1. Выводимость «снизу вверх» и «сверху вниз» 229
8.2. Подпрограммы как методы (routines as features) 230
8.3. Инкапсуляция (скрытие) функциональной абстракции 231
8.4. Анатомия объявления метода 232
8.5. Скрытие информации 234
8.6. Процедуры против функций 236
8.7. Функциональная абстракция 236
8.8. Использование методов 237
8.9. Приложение: Доказательство неразрешимости проблемы остановки 238
8.10. Дальнейшее чтение 240
8.11. Ключевые концепции, изученные в этой главе 240

9 Переменные, присваивание и ссылки 242

9.1. Присваивание 243
9.2. Атрибуты 250
9.3. Виды компонентов класса (features) 256
9.4. Сущности и переменные 260
9.5. Ссылочное присваивание 262
9.6. Программирование со ссылками 265
9.7. Ключевые концепции, изучаемые в этой главе 276

Часть II Как все устроено 279

10 Немного об аппаратуре 279

10.1. Кодирование данных 280
10.2. О памяти 287

10.3. Команды компьютера 292
10.4. Закон Мура и эволюция компьютеров 294
10.5. Дальнейшее чтение 294
10.6. Ключевые концепции, изученные в этой главе 295

11 Описание синтаксиса 298

11.1. Роль БНФ 298
11.2. Продукции 302
11.3. Использование БНФ 306
11.4. Описание абстрактного синтаксиса 309
11.5. Превращение грамматики в анализатор 311
11.6. Лексический уровень и регулярные автоматы 311
11.7. Дальнейшее чтение 318
11.8. Ключевые концепции, изучаемые в этой главе 318

12 Языки программирования и инструментарий 320

12.1. Стили языков программирования 321
12.2. Компиляция против интерпретации 329
12.3. Основы компиляции 334
12.4. Верификация и проверка правильности 340
12.5. Текстовые редакторы на этапах проектирования и программирования 340
12.6. Управление конфигурацией 342
12.7. Репозиторий проекта «в целом» 349
12.8. Просмотр и документирование 349
12.9. Метрики 350
12.10. Интегрированная среда разработки 350
12.11. IDE: EiffelStudio 351
12.12. Ключевые концепции, введенные в этой главе 356

Часть III Алгоритмы и структуры данных 358

13 Фундаментальные структуры данных, универсальность и сложность алгоритмов 358

13.1. Статическая типизация и универсальность 359
13.2. Контейнерные операции 365
13.3. Оценка алгоритмической сложности 370
13.4. Массивы 373
13.5. Кортежи 381
13.6. Списки 383
13.7. Связные списки 391
13.8. Другие варианты списков 398
13.9. Хеш-таблицы 401
13.10. Распределители 407
13.11. Стеки 409
13.12. Очереди 416
13.13. Итерирование структуры данных 418
13.14. Другие структуры 420
13.15. Дальнейшее чтение 420

13.16. Ключевые концепции этой главы 421

14 Рекурсия и деревья 423

14.1. Основные примеры 424
14.2. Ханойская башня 429
14.3. Рекурсия как стратегия решения задач 433
14.4. Бинарные деревья 434
14.5. Перебор с возвратами и альфа-бета 446
14.6. От циклов к рекурсии 455
14.7. Понимание рекурсии 457
14.8. Контракты рекурсивных программ 466
14.9. Реализация рекурсивных программ 468
14.10. Ключевые концепции, рассмотренные в этой главе 479

15 Проектирование и инженерия алгоритма: топологическая сортировка 484

15.1. Постановка задачи 484
15.2. Основы топологической сортировки 487
15.3. Практические соображения 495
15.4. Базисный алгоритм 502
15.5. Уроки 515
15.6. Ключевые концепции этой главы 518
15.7. Приложение. Терминологические замечания об отношениях порядка 519

Часть IV Объектно-ориентированные приемы программирования 522

16 Наследование 522

16.1. Такси и транспортные средства 524
16.2. Полиморфизм 528
16.3. Динамическое связывание 532
16.4. Типизация и наследование 533
16.5. Отложенные классы и компоненты 535
16.6. Переопределение 540
16.7. За пределами скрытия информации 543
16.8. Беглый взгляд на реализацию 544
16.9. Что происходит с контрактами? 549
16.10. Общая структура наследования 554
16.11. Множественное наследование 555
16.12. Универсальность плюс наследование 561
16.13. Обнаружение фактического типа 565
16.14. Обращение структуры: посетители и агенты 571
16.15. Дальнейшее чтение 578
16.16. Ключевые концепции этой главы 578

17 Операции как объекты: агенты и лямбда-исчисление 583

17.1. За пределами двойственности 583

17.2. Зачем операции превращать в объекты? 584
17.3. Агенты для итерации 590
17.4. Агенты для численного интегрирования 597
17.5. Открытые операнды 599
17.6. Лямбда-исчисление 602
17.7. Манифестные агенты, непосредственно определяющие функцию 613
17.8. Конструкции в других языках 614
17.9. Дальнейшее чтение 617
17.10. Ключевые концепции данной главы 617

18 Проектирование, управляемое событиями 621

18.1. Управляемое событиями GUI-программирование 622
18.2. Терминология 624
18.3. Требования «Публиковать-подписаться» 630
18.4. Образец «Наблюдатель» 634
18.5. Использование агентов: библиотека EVENT 641
18.6. Дисциплина подписчиков 644
18.7. Архитектура ПО. Уроки 645
18.8. Дальнейшее чтение 649
18.9. Ключевые концепции, изученные в этой главе 650

Часть V Цель – инженерия программ 652

19 Введение в инженерию программ 652

19.1. Базисные определения 653
19.2. DIAMO – облик инженерии программ 654
19.3. Составляющие качества 655
19.4. Главные виды деятельности в процессе разработки ПО 661
19.5. Модели жизненного цикла и разработка в стиле AGILE 663
19.6. Анализ требований 666
19.7. В&П – Верификация и проверка правильности 674
19.8. Модели способностей и зрелости 679
19.9. Дальнейшее чтение 683
19.10. Ключевые концепции, изложенные в этой главе 686

Часть VI Приложения 688

A Введение в Java (по материалам Марко Пиккони) 688

A.1. Основы языка и стиль 688
A.2. Общая структура программы 689
A.3. Базисная ОО модель 691
A.4. Наследование и универсальность 700
A.5. Другие механизмы структурирования программ 702
A.6. Отсутствующие элементы 705
A.7. Специфические свойства языка 706
A.8. Лексические и синтаксические аспекты 711
A.9. Библиография 712

B Введение в C# (по материалам Бенджамина Моранди) 713

B.1. Окружение языка и стиль 714
B.2. Общая структура программы 715
B.3. Базисная ОО-модель 716
B.4. Наследование 729
B.5. Другие механизмы структурирования программ 734
B.7. Специфические свойства языка 737
B.8. Лексические аспекты 738
B.9. Библиография 738

C Введение в C++ (по материалам Надежды Поликарповой) 739

C.1. Основы языка и стиль 739
C.2. Общая организация программ 740
C.3. Базисная ОО-модель 742
C.5. Дополнительные механизмы структурирования программ 760
C.6. Отсутствующие элементы 761
C.7. Специфические свойства языка 762
C.8. Библиотеки 762
C.9. Синтаксические и лексические аспекты 763
C.10. Дальнейшее чтение 768

D От C++ к C 769

D.1. Отсутствующие элементы 769
D.2. Основы языка и стиль 770
D.3. Дальнейшее чтение 771

E Использование среды EiffelStudio 772

E.1. Основы EiffelStudio 772
E.2. Запуск проекта 773
E.3. Показ классов и обликов 774
E.4. Как задать корневой класс и процедуру создания 774
E.5. Управление контрактами 774
E.6. Управление выполнением и инспекция объектов 775
E.7. Состояние паники (ни в коем случае!) 775
E.8. Узнать больше 775

Доступные ресурсы

«Почувствуй класс» основан (на момент публикации оригинала книги) на шестилетнем опыте преподавания курса «Введение в программирование» в ETH (Цюрих), который читается всем студентам, поступающим на отделение информатики (computer science). Параллельно с созданием курса и книги разработан большой объем учебных материалов. Приветствуется использование этих материалов при обучении студентов.

На сайте, посвященном курсу и книге:

http://touch.ethz.ch,

можно найти ссылки на:

- полный набор слайдов к курсу (PowerPoint + PDF) в его последней версии;
- доступные для загрузки видеозаписи лекций;
- дополнительные материалы;
- упражнения;
- слайды для практических занятий (учебные пособия);
- страницу Wiki для преподавателей, использующих эту книгу в качестве учебника;
- доступную для загрузки программную систему «*Traffic*» (Windows, Linux...);
- опубликованные статьи и отчеты о нашей учебной деятельности, которая связана с курсом и другими работами, ведущимися на факультете, включая разработку основ курса TrucStudio;
- информацию о курсах в других университетах, применяющих эту книгу как учебник;
- список опечаток;
- уголок преподавателей (требуется регистрация) — активных членов сообщества, обменивающихся опытом работы.

Все материалы находятся в свободном доступе для академического использования (условия лицензирования смотри на сайте). Пожалуйста, свяжитесь с нами, если предполагаются другие формы работы.

Большинство материалов, в частности, слайды и видеозаписи, сделаны на английском. Доступна также версия упражнений и слайдов на немецком языке. Мы надеемся на появление материалов и на других языках, предоставленных преподавателями разных стран. Будем рады включить в наш сайт переводы слайдов и других материалов.

Добро пожаловать в наше сообщество!

Посвящение

Эта книга посвящена двум пионерам информатики в знак благодарности за их неоценимое влияние и блестящее понимание сути вещей:

Энтони Хоару (*C.A.R. Hoare*) – по случаю его 75-летия и Никласу Вирту (*N. Wirth*) – с особой благодарностью за его вклад в становление информатики в ETH.

Предисловие редактора перевода

Так учат в ЕТН

Швейцарское федеральное высшее техническое училище – ETH (Eidgenossische Technische Hochschule) входит в пятерку лучших университетов Европы. Для программистов ЕТН славен тем, что именно здесь Никлас Вирт создал знаменитую серию языков программирования – Algol-W, Pascal, Modula, Oberon. Язык Pascal воплотил идеи структурного программирования и структурных типов данных, став де-факто языком, на котором в большинстве университетов учили программированию в течение десятилетий.

Последние 8 лет кафедру Вирта Software Engineering возглавляет профессор Бертран Мейер (по правилам Швейцарии профессор Вирт после 65 лет не занимает административных постов, оставаясь почетным профессором и научным консультантом университета). Меняются персоналии, но традиция остается, и эта кафедра ЕТН по-прежнему остается центром, генерирующим новые идеи, как в программировании, так и в обучении программированию.

Перед нами фундаментальный труд профессора Бертрана Мейера, создававшийся в течение 6 лет преподавания в ЕТН. Книга, рассказывающая о том, чему и как учат программистов в ЕТН, интересна, полезна, необходима всем, кто учит и учится программированию. Это первый и очевидный круг читателей книги.

Если бы эта книга была просто учебником по программированию, ориентированная только на студентов и преподавателей вузов, то и в этом случае ей цены бы не было. Но эта книга, как и другие книги Бертрана Мейера, интересна всем работающим программистам вне зависимости от уровня их квалификации. Она интересна уже тем, что учит не просто программированию, она учит, как программировать ***хорошо***. В ней, помимо важных аспектов информационных технологий есть нечто большее: она учит философии мышления человека, создающего программные продукты.

Достаточно трудно определить суть программирования как научной дисциплины. Известная книга Кнута называется «Искусство программирования», книга Гриса – «Наука программирования», книга Соммервилла – «Инженерия программного обеспечения». Википедия утверждает, что программирование содержит все эти части – науку, искусство и инженерию.

Книга Бертрана Мейера «Почувствуй класс» дает нам прекрасный пример сочетания этих разных сторон программирования. Здесь есть хорошая математика, например, приводятся разные варианты доказательства фундаментальной теоремы «о неразрешимости проблемы остановки». Глава 15, посвященная разработке алгоритма и программы топологической сортировки, – это образец искусства программирования. Рефрен всей книги – инженерия программ. Автор стремится с первых шагов учить профессиональному стилю программирова-

ния, ориентированному на создание программных продуктов высокого качества, создаваемого из повторно используемых компонентов, имеющих долгую жизнь.

И при обучении, и в реальной работе крайне важно и крайне сложно соблюсти баланс между этими гранями программирования. Принципиальное решение этой проблемы дает использование контрактов при проектировании и разработке программной системы. «Проектирование по контракту» – это главный вклад профессора Бертрана Мейера в современное программирование.

Бертран Мейер – автор языка Eiffel и научный руководитель созданной им фирмы Eiffel Software, успешно работающей многие годы, реализовавшей многие крупные проекты в различных областях – здравоохранении, банковском секторе, обороне. Бертран Мейер не без основания полагает, что объектно-ориентированный язык Eiffel является лучшим языком как для целей обучения, так и для разработки серьезных промышленных проектов. Понятно, что язык Eiffel и среда разработки Eiffel Studio стали тем рабочим окружением, на базе которого строится обучение в ETH.

Несколько важных принципов положено в основу обучения:

- Начинать учить нужно сразу объектно-ориентированному программированию.
- С первых шагов студенты должны работать в мощной программной среде с множеством классов, создавая из готовых компонентов эффективные приложения с графическим интерфейсом (студентам ETH предоставляется специальная система Traffic, а также все библиотеки, используемые в Eiffel Studio).Такой подход называется «обращенным учебным планом».
- Для работы в такой среде достаточно знания интерфейсов, построенных на контрактах. У студентов с самого начала вырабатывается понимание важности спецификации разрабатываемого ПО. Код всего программного обеспечения, предоставляемого студентам, открыт, что позволяет перейти на нужном этапе от понимания интерфейса к пониманию реализации. Такой подход называется «извне – внутрь».

Возникает естественный вопрос, насколько опыт обучения в ETH может быть тиражирован и использован при обучении в других университетах, в частности в России? Заметьте, все программные средства, все учебные материалы ETH доступны для свободного использования всеми заинтересованными преподавателями..

Не хочу выступать в роли апологета Eiffel и призывать всех завтра же учить так, как учат в ETH сегодня (полагаю, что найдутся те, кто, прочитав эту книгу, решится на такой шаг).

Но вот что совершенно бесспорно. Опыт обучения программированию в ETH нельзя не учитывать. В первую очередь это касается контрактов. Они должны появляться с первых программ, написанных студентами, они должны быть неотъемлемой частью профессионально разрабатываемого ПО. Хороший курс обучения программированию, так же, как и курс, реализованный в ETH, должен включать все грани программирования – науку, искусство и инженерию.

Я уже говорил, что признанных учебников по программированию, используемых в разных университетах, до сих пор нет. Книга «Почувствуй класс. Учимся программировать хорошо с объектами и контрактами» – один из претендентов на эту роль.

Как всегда при переводе возникали проблемы с переводом тех или иных терминов. Как правило, для большинства терминов при первом упоминании в скобках указывается оригинальное значение термина. Некоторые пояснения по поводу перевода отдельных терминов даются в сносках по ходу изложения. Все страничные сноски принадлежат редактору перевода.

В заключение несколько слов о профессоре Бертране Мейере. Вот что говорится о нем в Википедии: «Бертран Мейер является одним из ведущих ученых в области инженерии программного обеспечения. Он автор девяти книг. Им опубликовано более 250 научных работ,

охватывающих широкий спектр направлений, все из которых трудно перечислить. Вот лишь некоторые из них: методы построения надежных, повторно используемых компонентов и программных продуктов, параллельное, распределенное и интернет-программирование, технологии баз данных, формальные методы и доказательство корректности программ».

Он хорошо известен российским программистам. Почетный доктор СПбГУ ИТМО. Из 9 его книг это – третья книга, переведенная на русский язык. Первая книга, «Методы программирования», появилась еще в 1982 году под редакцией Андрея Петровича Ершова. Перевод второй его книги, «Объектно-ориентированное конструирование программных систем», называемой без преувеличения библией «объектно-ориентированного программирования», вышел в 2005 году с большим запозданием по отношению к первому изданию оригинала книги. Оригинал данной книги, «Touch of Class. Learning to Program Well with Objects and Contracts» вышел в 2009 году, так что запаздывание с переводом не столь велико.

Бертран Мейер знает русский язык, в молодые годы проходил стажировку у А. П. Ершова в Академгородке, где была написана одна из первых его статей. Довольно часто приезжает в Россию, выступает на конференциях. Последняя организованная им международная конференция по программной инженерии – SEAFOOD 2010 – проходила в июне 2010 года в Санкт-Петербургском государственном университете.

Нельзя не отметить стиль, в котором написана эта книга. Как и все книги Бертрана Мейера, она интересна своим широким культурным контекстом. Символично то, что на обложке оригинала книги «Touch of Class» помещена картина Рафаэля «Академия Платона», в центре которой Платон, беседующий с учениками.

Позволю и себе закончить предисловие несколькими литературными ассоциациями. Эпиграфом к книге вполне могли бы служить строки стихотворения Бориса Пастернака:

Во всем мне хочется дойти
До самой сути.
В работе, в поисках пути,
В сердечной смуте.

До сущности протекших дней,
До их причины,
До оснований, до корней,
До сердцевины.

В книгах Бертрана Мейера никогда читателю не дается готовый результат. Все изложение представляет исследование, в котором автор вместе с читателем пытается дойти до самой сути вещей. Книгу можно сравнить с романом Льва Толстого «Война и мир», признанным лучшим произведением мировой литературы. Хотя я знаю тех, кто не смог одолеть начальных страниц этого романа, насыщенных французской речью. Конечно, можно жить, не прочитав «Войны и мира», можно быть программистом, «не почувствовав класс», но стоит помнить, что великие книги нужно читать.

Надеюсь, что читатели новой книги Бертрана Мейера получат наслаждение от понимания того, как *программировать хорошо с объектами и контрактами*.

Владимир Биллиг

Предисловие автора к русскому изданию

Книга «Почувствуй класс» представляет новый подход в обучении началам программирования — подход, доказавший на практике свою успешность и применяемый в ETH Цюрих вот уже восьмой год.

Книга написана в первую очередь для тех студентов, кто выбрал информатику (computer science) своей специальностью, и использовалась не только в ETH, но и в других университетах. Лекторы, желающие применить книгу при обучении, могут найти много полезных ресурсов (слайды презентаций, упражнения, видеозаписи лекций, список опечаток, форум для преподавателей и прочее), регулярно обновляемых на сайте http://touch.ethz.ch.

Публикация английского издания показала, что многие другие читатели, включая инженеров и менеджеров, работающих в индустрии, находят полезным *«Почувствуй класс»*, открывая для себя источники новых идей современного программирования. Еще одна категория читателей — специалисты в дисциплинах, отличных от информатики, стремящиеся понять основы концепции программирования, не ограничиваясь техническими деталями.

Сегодня нельзя учить программированию так, как учили нас и многие поколения программистов (в деталях это обсуждается в основном предисловии к книге). Причина не только в том, что изменилось программирование, которое стало столь вездесущим, и не только в том, что программы стали столь сложными, но и в том, что изменились студенты. Сегодня на начальный курс редко приходят студенты, вовсе не знакомые с программированием. Из них примерно 85% уже имеют программистский опыт, и примерно 15% уже создавали довольно большие программы. Что же они ожидают от университетского курса, и что мы должны дать им? Как сказано в заглавии книги, нужно учить их программировать *хорошо*. Этим они будут отличаться от миллионов людей, знакомых с программированием, но не являющихся профессионалами высокого уровня. Цель этой книги – направить читателей по пути, ведущему к становлению профессионалов.

«Почувствуй класс» покрывает широкий спектр тем, часть из которых — довольно продвинутые и обычно не включаются в начальный курс. Наш подход к обучению «извне — внутрь», подробно описанный в большом предисловии, основан на свободно распространяемой библиотеке классов Traffic, специально построенной для этих целей. Он позволяет читателям учиться на примерах и одновременно является источником вдохновения для собственных разработок. Полагаю, что это лучший способ стать первоклассным программистом. Я настоятельно рекомендую преподавателям, использующим книгу при обучении, давать студентам упражнения, применяя Traffic и другое ПО с открытым кодом, поддерживающее этот курс.

«Почувствуй класс» имеет четкую ориентацию на практику работы, но равновесно содержит серьезный теоретический материал, включающий логические утверждения, сложность алгоритмов, лямбда-исчисление. Центральной темой, объединяющей процесс обучения, яв-

ляется тема «Проектирования по Контракту». Это единственный путь, насколько я знаю, создавать высококачественное ПО, функционирующее корректно, устойчиво и безопасно. Не следует откладывать изучение этих методов на будущее в углубленных курсах по программированию. Они могут применяться всеми программистами для любых приложений, и умение ими пользоваться — еще один пример того, что отличает профессионала от любителя.

Наш подход — полностью объектно-ориентированный с акцентами на скрытие информации и проектирование интерфейсов, с рассмотрением таких приемов, как универсальность, наследование (одиночное и множественное), агенты (замыкания или делегаты). Эти механизмы становятся центральными в современном программировании, поэтому они могут и должны изучаться в начальном курсе.

***

В последние годы я установил тесные отношения как с рядом университетов России, так и с другими организациями (или, точнее, восстановил ранее существовавшие связи). Я высоко оцениваю качество компьютерных наук в России, и потому меня радует появление русского издания «Почувствуй класс».

Мне особенно приятно, что перевод этой книги сделан профессором Владимиром Биллигом. Его предыдущий перевод моей книги «Объектно-ориентированное конструирование программных систем» (Русская Редакция, 2000) заслужил много комплиментов за стиль и высокое качество. Владимир Биллиг многократно помогал мне, когда я готовил доклады на русском языке и выступал с ними в России. Те же навыки и талант он продемонстрировал и при переводе *«Почувствуй класс»* (обнаружив в процессе перевода ряд опечаток в английской версии, которые скорректированы в русском издании). Благодаря его интенсивной работе русская версия появляется практически вслед за английским изданием. Я благодарен профессору Биллигу за сотрудничество, продолжающееся уже многие годы.

Обращаясь к российским читателям, хочу выразить надежду, что, будь они студенты или профессионалы, книга *«Почувствуй класс»* покажется им интересной и полезной.

Бертран Мейер,
Цюрих, декабрь 2010

Предисловия

```
note
    description:"[
        В этой книге два предисловия: одно для преподавателей, другое - для студен-
        тов, что отражено ниже в непривычной, но корректно используемой нашей собст-
        венной нотации, применяемой при записи программ.
                    ]"
class PREFACING inherit
    KIND_OF_READER
create
    choose
feature — Инициализаци¤
    choose
            — Выбрать предисловие, предназначенное для вас.
        do
            if is_student then
                student_preface.read
            elseif is_instructor then
                instructor_preface.read
            else
                pick_one_or_both
            end
                check
                    — Вы узнаете о динамическом связывании.
                note
                    why: "Вы сумеете выразить все элегантнее!"
                end
        end
end
```

Предисловие для студентов

Программирование – увлекательное занятие (fun)! Где еще можно проводить дни, создавая машины собственным воображением, строя их без молотка и гвоздей, не пачкая одежды, заставляя их работать как по волшебству, и при этом ежемесячно получать плату – довольно неплохую? (Спасибо за существующий спрос на такую работу!)

Программирование – занятие сложное! Где еще продукты от самых престижных компаний отказывают в обычных условиях использования? Где еще вы найдете так много возмущенных пользователей? Где еще инженеры проводят часы и дни в попытках понять, почему то, что должно работать, не работает?

Будьте готовы к овладению мастерством программирования в его профессиональной форме – инженерии программ (software engineering); будьте готовы к радостям и трудностям этой профессии.

Программное обеспечение (ПО, software) повсюду

Выбрав информатику, вы выбрали одну из наиболее захватывающих и быстро развивающихся областей науки и техники. Пятьдесят лет назад едва ли можно было говорить об информатике как научной отрасли; сегодня невозможно представить университет без факультета информатики. Тысячи книг и журналов появляются в этой области, проводятся тысячи конференций. Общие доходы в индустрии, называемой «Информационные технологии» (ИТ), измеряются триллионами долларов. Нет другой отрасли в истории технологий, растущей с такой скоростью и значимостью.

Дело в том, что без ПО невозможны были бы межконтинентальные перелеты, и фактически не было бы и самих современных лайнеров (а также и современных автомобилей, скоростных поездов и прочего), так как их проектирование требует весьма сложного ПО, называемого CAD (Computer-Aided Design).

Чтобы платить зарплату своим служащим, любая большая компания должна была бы иметь сотни работников бухгалтерии, занятых начислением и выписыванием расчетных чеков. Телефон до сих пор висел бы на стене с протянутыми к нему проводами. Сделав снимок фотоаппаратом, вы не смогли бы увидеть результат, не проявив пленку и не напечатав фотографии. Не было бы ни видеоигр, видеокамер, iPods и iPhones, ни Skype, ни GPS. Любой отчет нужно было бы написать вручную и отдать машинистке для перепечатывания, а потом править полученный текст, повторяя этот процесс до получения желаемого результата.

Неожиданное желание узнать имя капитана-артиллериста в романе «Война и мир» или численность населения Кейптауна, или автора известного высказывания, – все это требовало бы путешествия в библиотеку. Теперь же достаточно в окне поиска напечатать несколько слов и мгновенно получить ответ.

Этот список новых возможностей, составляющих нашу повседневную жизнь, можно продолжить. В их основе лежат программы – невероятно сложные программы.

Все это не случилось само по себе, по мановению волшебной палочки. Программирование, задача конструирования новых программ или улучшения существующих, является интеллектуальным занятием, требующим созидательного мышления и опыта. Эта книга поможет вам войти в мир программ и программирования, стать профессионалом в этой области.

Любительская и профессиональная разработка ПО

Все больше и больше людей получают знания по основам информатики, но профессиональное программирование предполагает совершенно иной уровень мастерства.

Для сравнения рассмотрим математику. Несколько столетий назад способность складывать и вычитать пятизначные числа требовала университетского образования и обеспечивала возможность получения хорошей работы в качестве бухгалтера. Сегодня этим навыкам учат в средней школе. Если вы хотите стать инженером, физиком или биржевым игроком, вам необходимо изучить в университете более продвинутые разделы математики, такие как, например, дифференциальное и интегральное исчисление. Граница между базисными знаниями и университетским уровнем образования существенно сдвинулась в сторону усложнения.

Компьютерная обработка информации, то, что чаще называют одним словом – компьютинг (Computing), следует тем же путем, но со значительно большей скоростью: счет идет на десятилетия, а не на столетия, как ранее. Еще недавно умения написать программу для компьютера было достаточно для получения хорошей работы. В наши дни вы никого не удивите, указав в резюме «Я пишу программы», – это все равно, как если бы вы указали, что умеете складывать числа.

Есть существенная разница между базисными программистскими навыками и квалификацией специалиста в программной инженерии. Основы будут доступны всем получившим современное образование. Профессиональное образование подобно продвинутым разделам математики. Изучение этой книги – шаг в направлении, ведущем к профессионалам компьютинга.

Факторы, отличающие профессиональное программирование от любительского, включают **объем**, **длительность** и **изменения**. В профессиональной разработке приходится иметь дело с программами, содержащими миллионы строк кода, работающими в течение нескольких лет или десятилетий, подверженными многочисленным изменениям и расширениям в ответ на изменившиеся обстоятельства. Многие проблемы кажутся незначительными и тривиальными при работе с программами средних размеров, но они становятся критическими при переходе к профессиональному программированию.

В этой книге я буду пытаться подготовить Вас к миру реального ПО, где системы сложны, где решаются серьезные проблемы, зачастую критически важные для здоровья человека или его собственности, к миру, где программы живут долго и должны приспосабливаться к изменениям.

Предшествующий опыт

Эта книга не предполагает наличия предшествующего опыта и программистских знаний.

Если вы уже писали программы, то этот опыт поможет вам быстрее овладеть концепциями. Вы будете знакомы с некоторыми идеями, но должны быть готовы к тому, что времена-

ми вам предстоит удивляться: профессиональное изучение отличается от общего пользовательского опыта. Например, вам может показаться, что я напрасно растолковываю, казалось бы, простую вещь. Если так, то вскоре, я надеюсь, вы обнаружите, что не все так просто, как кажется с первого взгляда. Для математика сложение — вещь куда более тонкая, чем для счетовода.

Вы можете и должны воспользоваться всеми преимуществами того, что вы уже знаете, но приготовьтесь, что применяемые ранее приемы не будут соответствовать принципам изучаемой здесь программной инженерии. Изучение программирования требует много усилий: каждый кусочек, каждый аспект, помогающий приблизиться к пониманию, полезен.

В данной книге обсуждение строится, как будет пояснено ниже, на поддерживающей обучение программной системе Traffic. Если вы знакомы с программированием, вы можете обнаружить некоторые возможности системы самостоятельно, помимо официальных заданий. Не раздумывая, так и поступайте: программированию учатся на примерах, изучая существующие программы и создавая собственные версии. Возможно, вам придется сделать некоторые предположения об элементах Traffic, которые основаны на приемах и конструкциях языка, еще не изученных вами, но и тогда имеющийся опыт может помочь вам двигаться быстрее.

С другой стороны, если у вас *нет* опыта программирования, это тоже не страшно. Возможно, прогресс на первых порах не будет быстрым, вам придется тщательнее изучать все материалы и выполнять все упражнения. Все это верно и по отношению к математике. Хотя эта книга включает не так много настоящей математики, вы будете чувствовать себя комфортнее, если обладаете математическим мышлением и навыками логического вывода. Это так же полезно, как и программистский опыт, и компенсирует ту фору, которую имеют ваши сокурсники, позиционирующие себя так, как будто они программируют с тех пор, как у них прорезались молочные зубы.

Программирование, подобно другим направлениям информатики, является смесью инженерии и науки. Успех требует как владения практическими приемами («хакерская» сторона дела в позитивном смысле этого слова), полезными в технологически-ориентированной работе, так и способности строить абстрактные логические выводы, требуемые в математике и других науках. Программистский опыт помогает в достижении первой цели, математическое мышление – второй. Используйте ваши сильные стороны, используйте эту книгу, позволяющую ликвидировать ваше начальное отставание.

Современная технология ПО

Для того чтобы стать профессионалом, вам потребуется не один курс и не одна книга. Требуется многолетний учебный план, в котором, помимо математических основ, таких как логика и математическая статистика, изучаются теория трансляции, структуры данных, теория алгоритмов, операционные системы, искусственный интеллект, базы данных, вычислительная техника, компьютерные сети, управление проектом, вычислительная математика, программные метрики, графика и многие другие дисциплины. Но чтобы быть готовыми воспринять основы этих курсов, требуется хорошее знание технологии ПО.

В последние годы две главные идеи, составляли потенциал создания качественного ПО — **объектно-ориентированное конструирование ПО** (ОО ПО) и **формальные методы**. Обе эти идеи, особенно первая, делают изучение введения в компьютинг более захватывающим и полезным. Они же, наряду с другими концепциями современной технологии разработки ПО, играют главную роль в этой книге. Давайте начнем наше первое беглое знакомство с этими идеями.

Объектно-ориентированное конструирование ПО

В предыдущей книге (на русском языке – «Объектно-ориентированное конструирование ПО», изд. «Русская Редакция», Интернет-университет, 2005 г.) объектная технология рассматривалась на более глубоком уровне.

Конструирование ОО ПО исходит из осознания того, что правильно построенные системы программной инженерии должны основываться на повторно используемых компонентах высокого качества, как это делается в электронике или строительстве. ОО-подход определяет, какую форму должны иметь эти компоненты: каждый из них должен быть основан на некотором *типе объектов*. Термин «объект», давший имя этому методу, предполагает, что объектами являются не только сущности некоторой предметной области, такие как, например, круги и многоугольники в графической системе, но также объекты, имеющие чисто внутренний для системы смысл, например, списки. Если вы не совсем понимаете, что все это значит, – это нормально. Мы надеемся, что, если вы перечитаете это введение несколько месяцев спустя, все будет кристально понятно!

Объектная технология (краткое имя для ОО-конструирования ПО) быстро изменила индустрию ПО. Овладение ей с первых шагов изучения компьютинга – лучшая страховка от технической отсталости.

Формальные методы

Формальные методы являются приложением систематических способов доказательства, основанных на математической логике, к конструированию надежного ПО. Надежность, а скорее отсутствие ее, – одна из самых неприятных проблем ПО. Ошибки или страх ошибок служат постоянным фоном программирования. Всякий, кто использует компьютеры, знает дюжину анекдотов по поводу тех или иных «багов».

Формальные методы могут помочь улучшить ситуацию. Изучение формальных методов в полном объеме требует больших знаний, чем это доступно для начального уровня университетского образования. Но подход, используемый в этой книге, показывает важность влияния формальных методов, в частности, благодаря «*Проектированию по Контракту*» (*Design by Contract*), в котором конструирование ПО рассматривается как реализация некоторых контрактных отношений, существующих между модулями программной системы. Каждый контракт характеризуется точной спецификацией прав и обязанностей. Надеюсь, вы поймете важность этих идей и будете помнить их на протяжении всей вашей карьеры. В индустрии хорошо известна разница между «кодировщиком» и программистом, способным создавать корректные, устойчивые программные элементы длительного пользования.

Учимся, создавая

Эта книга – не теоретическая презентация. Она предполагает практику работы параллельно с изучением теоретических материалов. Ассоциированный веб-сайт – touch.ethz.ch – предоставляет ссылки на необходимое ПО в версиях для Windows, Linux и других платформ, свободное для загрузки. В некоторых случаях практическая работа с ПО предваряет изучение теоретических концепций.

Система, используемая в этом курсе, содержит развитое ОО-окружение – EiffelStudio, среду разработки программных систем на языке Eiffel, применяемом на этапах анализа,

проектирования и программирования. Язык Eiffel – это простой, современный язык, широко используемый в мире для больших критически важных индустриальных проектов в таких областях, как банковское дело, здравоохранение, сетевые разработки, аэрокосмическая индустрия. Этот язык также широко применяется в университетах при обучении и исследованиях. Задействованная при обучении версия EiffelStudio немногим отличается от профессиональной версии — у нее тот же графический интерфейс и те же базисные повторно используемые компоненты, представленные в библиотеках: EiffelBase, EiffelVision и EiffelMedia.

Ваш университет может получить академическую лицензию, обеспечивающую поддержку и сопровождение данного ПО.

Четыре других языка программирования, широко используемые в индустрии: Java, C#, C++ и C — рассмотрены в приложениях к данной книге.

Любой профессиональный программист должен свободно владеть несколькими языками, по крайней мере, некоторыми из упомянутых. Изучение Eiffel будет несомненным плюсом в вашем резюме (признак профессионала) и поможет вам стать мастером и в других ОО-языках.

От потребителя к поставщику

Поскольку с первого дня курса вы будете иметь всю мощь EiffelStudio на кончиках ваших пальцев, существует возможность пропустить «детские» упражнения, традиционно используемые при обучении программированию. Наш подход основан на том наблюдении, что обучение ремеслу лучше всего начинать с изучения работ, выполненных профессионалами, получать преимущества, используя результаты этих работ, понимая их внутреннюю конструкцию, дополняя их, улучшая их и начиная строить свои собственные конструкции. Это проверенный временем почтенный метод ученичества, когда подмастерье работает под руководством мастера.

В роли изделия, созданного мастером, выступает специальная *библиотека классов* Traffic, разработанная для курса и этой книги. При написании первых программ вы будете использовать эти классы, что позволит получать результаты вашей работы – достаточно выразительные, хотя и не требующие написания большого кода. В вашей программе вы будете действовать как *потребитель* существующих компонентов. Это напоминает ситуацию, когда человек, умеющий водить машину, учится, чтобы стать инженером автомобильной промышленности. В процессе обучения мы наберемся храбрости, поднимем капот и посмотрим, как эти классы устроены. Позднее начнем писать расширения классов, улучшать их и создавать собственные классы.

Библиотека Traffic, как следует из названия, обеспечивает механизмы, относящиеся к движению в городе: автомобилей, трамваев, такси – а также пешеходов. Она обеспечивает графическую визуализацию, моделирование, определение и анимацию маршрутов и все прочее. Здесь появляется богатый источник различных приложений и расширений: можно писать видеоигры, решать оптимизационные задачи, испытывать новые алгоритмы.

Встроенные примеры используют Париж как один из самых популярных туристских объектов мира. Но возможна адаптация для любого другого города, так как вся локальная информация отделена от программ и задается в файле формата XML. Создав такой файл, можно работать с выбранным вами городом. Например, курс, изучаемый в ETH, рассматривает городскую трамвайную сеть Цюриха, заменяющую систему метро Парижа.

Абстракция

То, что в основе работы лежат существующие компоненты, имеет еще одно следствие, важное для вашего образования как инженера ПО. Программные модули, которые вы повторно используете, имеют существенные размеры с большим количеством встроенных в них знаний. Было бы довольно трудно применять их в собственных приложениях, если бы предварительно пришлось читать их полные тексты. Вместо этого вы будете основываться на описании их *абстрактных интерфейсов*, извлеченных из программных текстов и содержащих только ту информацию, которая необходима потребителю продукта. Извлечение абстрактного интерфейса выполняется автоматически программными механизмами, составляющими часть EiffelStudio. Абстрактный интерфейс дает описание цели программного модуля, описание выполняемой им функции, но не дает описание того, как эта функция реализуется. Он описывает, *что* делает модуль, но не *как* он это делает. В терминологии ПО абстрактный интерфейс называется *спецификацией* (*specification*) модуля. Он не включает *реализацию* (*implementation*) модуля.

Эта техника позволит вам освоить один из ключевых навыков профессионального разработчика ПО: *абстракцию*, означающую в данном случае способность отделять цель любой части ПО от реализации — от деталей, зачастую весьма обширных. Каждый профессор, читающий лекции по информатике, почти заклинаниями убеждает студентов в необходимости абстракции, имея на то все основания. Здесь тоже есть немалая толика заклинаний, но обнадеживает то, что пользу абстракций можно увидеть на примерах, постигая на опыте преимущества, предоставляемые повторно используемыми компонентами. Когда вы начнете строить свое собственное ПО, вам придется следовать тем же принципам, поскольку это единственный способ справиться с хаосом сложных программных систем.

Преимущества абстракции довольно конкретны. Вы почувствуете их с первых шагов. Первая программа, которую предстоит написать, будет содержать только несколько строчек, но результаты ее работы будут значимыми (анимация маршрута на карте города). Абстрактные спецификации модулей системы Traffic позволят нам без труда их использовать. Если бы для использования модулей пришлось анализировать их тексты, мы быстро утонули бы в океане деталей и не смогли бы получить никакого результата.

Спецификация вместо текста – это все, что необходимо потребителю ПО. Спецификация – это спасительный круг, позволяющий не сгинуть в море сложностей.

Цель: качество

Эта книга учит не только методам, но и методологии. В течение всего курса вы будете сталкиваться с принципами проектирования и правилами стиля. Иногда будет казаться, что проще написать программу, не придерживаясь никаких рекомендуемых правил. Действительно, такое возможно. Но правила методологии – это водораздел, отделяющий любительские программы от ПО промышленного качества. Первые иногда работают, иногда нет, вторые – должны всегда работать в соответствии со своими спецификациями. Именно такое ПО мы и хотим научиться создавать.

Вы должны применять эти правила не потому, что так написано в этой книге и так говорят ваши профессора, но потому, что мощь и скорость компьютеров усиливают любые недочеты, казалось бы, незначительные. От программиста требуется обращать внимание как на всю картину в целом[1], так и на отдельные детали. Это обеспечит хорошую страховку в буду-

[1] «Картина маслом», как говорил герой фильма «Ликвидация».

щей карьере. Программистов много, но их реальная дифференциация состоит в способности создания ПО высокого качества.

Не будьте глупцами, успокаивая себя выражениями «это только упражнение» или «это только маленькая программа»:

- упражнения – именно то место, где следует учиться наилучшим из возможных методов работы. Когда фирма «Аэробус» пригласит вас написать управляющее ПО для их следующего лайнера, учиться будет слишком поздно;
- считать программу «маленькой» — это, чаще всего, неоправданное ожидание. В индустрии многие «большие» программы начинались как программы небольшого размера, которые со временем разрастались, поскольку у пользователей хорошей программы появляются все новые идеи и запросы на расширение ее функциональности.

Так что следует применять одни и те же методологические принципы ко всем разрабатываемым программам, независимо от того, маленькие они или большие, учебные или реальные.

Такова цель этой книги: не просто дать основы инженерии программ и позволить получить опыт в привлекательной работе по созданию работающих программ, но показать красоту принципов, методов, алгоритмов, структур данных и других определяющих дисциплину методик. И, может быть, самая главная цель — привить чувство, заставляющее вас делать не просто хорошее ПО, а выдающееся, стремление создавать программы высочайше возможного качества.

БМ

Цюрих / Санта-Барбара, Апрель 2009

Предисловие для преподавателей

Уже в самом заголовке книги отражен ее дух: не просто научить программированию, а научиться «программировать *Хорошо*». Я пытаюсь направить студентов по правильной дороге, так, чтобы они могли радоваться программированию – без радости далеко не уйдешь – и достигли успеха в карьере; не просто получили первую работу, но обладали способностью отвечать на новые вызовы в течение всей жизни.

Для достижения этой цели книга предлагает инновационные идеи, детализируемые в этом предисловии.

- **Обращенный учебный план**, известный также как подход «извне-внутрь», основанный на большой библиотеке повторно используемых компонентов.
- Всепроникающее применение **ОО** и **управляемой модели** (model-driven) технологий.
- Eiffel и Проектирование по Контракту (Design by Contract).
- Умеренная доза формальных методов.
- Включение с самого начала подходов, свойственных **программной инженерии**.

Эти методики в течение нескольких лет применялись в ETH в курсе «Введение в программирование», который читался всем студентам, поступающим на отделение информатики. Книга «*Почувствуй класс*» построена на этом курсе и вобрала в себя его уроки. Это также означает, что преподаватели, использующие ее как учебник, могут применять все разработанные нами методические материалы: слайды, презентации лекций, материалы для самообучения, студенческие проекты, даже видеозаписи лекций.

Вызовы первого курса

Факультеты информатики во всем мире продолжают спорить, как лучше учить началам программирования. Это всегда было трудной задачей, но новые вызовы добавляют новые проблемы:

- адаптация к всевозрастающей конкуренции;
- идентификация ключевых знаний и навыков, которым следует учить;
- внешнее давление и модные увлечения;
- широкий разброс начальной подготовки студентов;
- выбор примеров и упражнений;
- введение в реальные проблемы профессионального ПО;
- введение формальных методов, не отпугивающих студентов.

Адаптация к всевозрастающей конкуренции. Готовя профессионалов с прицелом на будущее, мы должны учить надежным, долговечным навыкам. Недостаточно давать им непосредственно применяемые технологии, для которых в нашей глобальной индустрии всегда найдутся дешевые программисты, доступные повсюду.

Идентификация ключевых знаний и навыков, которым следует учить. Программирование больше не является редким, специализированным умением. Многие люди занимаются различными элементарными формами программирования, например, написанием макросов, разработкой веб-сайтов, используя Python или ASP.NET. Специалистам программной инженерии необходимо больше, чем умение программировать; они должны управлять разработкой ПО с профессиональной тщательностью и тем самым отличаться от программистов-любителей.

Внешнее давление и модные увлечения. Очень важно сохранять холодную голову, невзирая на моду и внешнее давление. Мода непременно присутствует в нашей отрасли, и это не всегда плохо – структурное программирование, объектная технология и проектирование по образцам были когда-то модными. Мы должны убедиться, что идея прошла проверку временем, прежде чем испытывать ее на наших студентах. Справиться с внешним давлением – семья, окружение студентов — бывает более трудно. От нас требуют, чтобы мы учили специфическим технологиям, рекламируемым, позволяющим получить хорошо оплачиваемую работу. Но что если это направление ошибочно, и четыре года спустя появится новое направление, а старое будет забыто, и потребуются люди, умеющие решать проблемы, а не обладающие специфическими знаниями? Это наша обязанность: служить настоящим интересам студентов и их семей, обучая их фундаментальным предметам, позволяющим не только получить первую работу, но и продолжить успешную карьеру.

Достаточно глупа навязчивая идея, что после обучения резюме должно быть заполнено модными словечками – из страха не получить работу. Мировой феномен состоит в том, что на протяжении десятилетий у хорошего разработчика ПО нет проблем с поиском хорошей работы. Нет конца новым захватывающим предложениям в нашей области, несмотря на все мрачные прогнозы, распространяемые средствами массовой информации, после схлопывания «интернет-пузыря» и опасений, что все работы ушли в Индию, Бангалор... Но важна квалификация. Люди, которые получили и *сохранили* работу, – это не узко мыслящие специалисты, обучение которых сводилось к заголовкам сегодняшнего дня; это компетентные разработчики, обладающие широким и глубоким пониманием информатики, владеющие многими дополнительными технологиями.

Широкий разброс начальной подготовки студентов. Различие в начальной подготовке студентов усложняет задачу. Среди студентов, пришедших на первую лекцию вводного курса, можно найти тех, кто редко пользовался компьютером, и тех, кто уже строил коммерческие сайты, и все разнообразие между этими полюсами. Что может сделать преподаватель в такой ситуации?

- Заманчиво: отобрать подготовленных студентов и учить только их, но тогда отсеиваются те, кто не имел возможности или склонности работать с компьютером. По моему опыту, среди них есть те, кто позже могут стать прекрасными исследователями благодаря склонности к абстракции и увлечению математикой приоритетно перед компьютерами.
- Не следует впадать и в другую крайность – всех поставить на нижний уровень и там начать работать. Нужен способ захватить и удерживать внимание более опытных студентов, позволяя им использовать и расширять достигнутые преимущества.

Есть надежда, что повторно используемые компоненты могут быть решением проблемы. Дав студентам доступ к библиотекам высокого качества, мы позволяем новичкам, благодаря абстрактным интерфейсам, воспользоваться преимуществами их функциональности, без

понимания внутреннего устройства. Более продвинутые и любопытные студенты могут, опережая других, начать копаться во внутренностях компонентов и использовать их как образцы для своих собственных программ.

Выбор примеров и упражнений. Для того чтобы так работать, необходимы **примеры высокого качества**. Студенты, живущие сегодня в мире, наполненном чудесами мультимедиа, ожидают, что они начнут строить нечто большее, чем программы из джентльменского набора: «Постройте седьмое число Фибоначчи». Мы должны соответствовать ожиданиям «Поколения Nintendo», не допуская, конечно, чтобы технологический блеск затмевал обучение профессиональным навыкам.

Вариант этой проблемы – так называемый феномен Google: «Найти и приклеить» («Google-and-paste»). Вы даете упражнение, требующее решения, скажем, объемом в 100 строк кода. Интернет-зависимые студенты быстро находят на каком-нибудь сайте код, решающий проблему, – с той лишь разницей, что найденное в Интернете решение содержит 10000 строк кода, поскольку в Интернете решается более общая задача. Конечно, можно запретить пользоваться Интернетом или отбраковывать найденные там решения. Но ведь «Найти и приклеить» – это одна из форм повторного использования, хотя и не совпадающая с рекомендуемыми в учебниках формами работы. Подход, применяемый в этой книге, делает шаг вперед. Мы не только одобряем повторное использование, но и обеспечиваем большое количество кода (150000 строк Eiffel к моменту написания этого текста) для повторного использования и для подражания. Код доступен в исходной форме и явно спроектирован как модель хорошего проекта и реализации. Повторное использование – одна из лучших практик, поддерживаемая курсом с самого начала, но в форме, согласованной с принципами программной инженерии и основанной на абстрактных интерфейсах и контрактах.

Введение в реальные проблемы разработки профессионального ПО. На университетском уровне подготовки специалистов по информатике или программной инженерии нельзя учить программированию «в малом». Мы должны готовить студентов к тому, с чем имеют дело профессионалы, — программированию «в больших размерах». Не все приемы, пригодные для малых размеров программ, масштабируемы. Сама природа академического окружения, особенно на начальном уровне, затрудняет подготовку студентов к реальным проблемам промышленного ПО. Разработка таких программных систем ведется большим коллективом, содержит много строк кода, ориентирована на различных пользователей, сопровождается в течение многих лет, подвержена многочисленным изменениям. Использование студентами предоставленных им больших библиотек в определенной степени снимает эти вопросы.

Забота о масштабируемости придает особую важность последней проблеме: **введение формальных приемов, не отпугивающих студентов**. Советы или приказы студентам использовать сокрытие информации, контракты и принципы инженерии программ – для начинающих могут восприниматься как бесполезные заклинания. Введение основанных на математике формальных приемов, таких как аксиоматическая семантика, может только расширить пропасть между студентом и преподавателем. Парадоксально: те студенты, кто уже немного программирует и, казалось бы, мог бы извлечь пользу из таких указаний и приемов, большей частью отвергают их, поскольку они знают по собственному опыту, что, по крайней мере, для небольших программ можно достичь приемлемого результата без всяких строгих правил. Лучший способ привить методологические принципы – это прагматично показать на деле, как это помогает сделать нечто, что в противном случае было бы немыслимо. Доверие к мощным библиотекам повторно используемых компонентов возникает тогда, когда они да-

ют возможность, например, построить программу большой выразительной силы с графикой и анимацией. С самого начала курса студенты могут создавать важные приложения, визуальные, с различными эффектами. Результат достигается благодаря повторному использованию компонентов, определяемых абстрактными интерфейсами. Но они никогда бы не продвинулись далее нескольких классов, если бы должны были предварительно читать весь код.

Вместо восхваления абстракции, сокрытия информации и контрактов лучше позволить студентам использовать эти приемы и осознать, что методики работают. Если идея спасла вас от проблем, вы не станете отвергать ее как бесплодный теоретический совет.

Извне – Внутрь: Обращенный учебный план

Порядок тем в курсе программирования традиционно идет снизу вверх: все начинается с введения переменных и присваивания, затем идут управляющие структуры и структуры данных, и далее двигаемся, если время позволяет, к принципам модульного программирования и методике структурирования больших программ.

Этот подход дает студентам хорошее практическое понимание процесса создания программ. Но он не пригоден для обучения концепциям конструирования программных систем, которыми должны владеть инженеры ПО для успеха в профессиональной работе. Способности создавать программы больше не достаточно: многие разработчики, не являющиеся профессионалами, успешно могут выполнять эту работу. Профессионала отличает владение системными навыками разработки и сопровождения больших и сложных программ, адаптируемых к новым потребностям и открытых для повторного использования компонентов системы. Начинать с элементов нижнего уровня, как это рекомендуется в стандарте учебного плана «CS1», возможно, не лучший путь.

Вместо традиционного порядка «снизу вверх» или «сверху вниз» эта книга предлагает порядок «**извне – внутрь**». Он исходит из предположения, что наиболее эффективный способ обучения программированию состоит в использовании хорошего существующего ПО, где определение «хорошее» предполагает высокое качество кода и, что не менее важно, программных *интерфейсов* (API). Обучение на образцах во многом сводится к имитации проверенных временем моделей.

Изначально студенты получают мощное ПО: множество библиотек, называемое Traffic, где верхний слой написан специально для этой книги, а базисные слои (структуры данных, графика, GUI, мультимедиа, анимация ...) широко используются в коммерческих приложениях. Исходный код всех библиотек открыт, позволяя тем самым рассматривать его как хранилище качественных моделей, допускающих имитацию их студентами. На практике, по крайней мере, на первых порах, студенты используют это ПО в своих программах через спецификации API, называемые также *контрактами* (*contract views*). Основываясь на контрактах, в которых информация абстрагирована от реального кода, студенты могут строить интересные приложения, даже когда написанная ими часть состоит из нескольких строчек, в которых вызываются модули библиотеки. Прогрессируя в процессе обучения, они начинают строить более сложные программы и идут «внутрь», понимая устройство вызываемых модулей – открывая и изучая устройство «черных ящиков». К концу курса они должны уметь самостоятельно строить такие библиотеки.

Результатом стратегии «Извне — Внутрь» является «Обращенный учебный план», когда студенты начинают как *потребители* готовых компонентов, становясь по мере обучения *производителями*. Это не означает игнорирование в обучении концепций и навыков нижнего уровня — в конечном счете, наша цель научить студентов заботиться обо всем, что требует-

ся при создании ПО, начиная от общей картины до мельчайших деталей. В нашем подходе большое внимание уделяется архитектурным навыкам, которыми зачастую пренебрегают в подходе «снизу – вверх».

Этот подход нацелен на то, чтобы студенты могли владеть ключевыми концепциями программной инженерии, в частности, *абстракцией*. В моей карьере в индустрии я многократно наблюдал, что главное качество, выделяющее профессионала, — это способность к абстракции, умение отделять главное от второстепенного, долговременные аспекты — от преходящих, спецификацию — от реализации. Все хорошие учебники ратуют за абстракцию, но результат этих призывов будет слабым, если студенты знакомы с программированием только как с набором небольших примеров на реализацию алгоритмов. Я также могу прочесть лекцию про абстракцию, но наиболее эффективно будет сослаться на пример, показав студентам, как можно создать выразительное приложение благодаря повторному использованию существующего ПО. Наше ПО является большим по академическим стандартам, изучение его исходного кода заняло бы месяцы. И все же студенты могут уже в первую неделю курса строить нетривиальные приложения, используя знание контрактов.

Абстракция здесь – не просто прекрасная идея, не уговоры быть хорошими и все делать правильно. Это единственный путь достижения желаемой цели, стоя на плечах гигантов. Студенты не нуждаются в рассуждениях о пользе абстракции и повторного использования, если с первых шагов и повседневно они приобретают основанный на контрактах опыт построения мощных приложений благодаря повторному использованию библиотек. Для них эти концепции становятся второй природой.

Обучение лучше, чем молитва, и если есть нечто лучше обучения, то это демонстрация принципов работы, осуществленная самими студентами. Сделайте так — и услышите «Вау!».

Поддерживающее ПО

Центральным для подхода, представленного в этой книге, является ПО Traffic, доступное для свободной загрузки на сайте *touch.ethz.ch*.

Выбор области приложения для поддерживающей библиотеки должен отвечать ряду требований.

- Предметная область должна быть знакома всем студентам так, чтобы они могли проводить время в изучении решений, хорошо понимая проблему. Было бы интересно рассмотреть, например, астрономию в качестве предметной области. Но тогда, боюсь, больше времени пришлось бы потратить на обсуждение комет и галактик, чем на наследуемые структуры и инварианты классов.
- Проблемная область должна обеспечивать большое множество интересных алгоритмов, примеров структур данных, применение фундаментальных концепций, позволять преподавателю создавать новые примеры, помимо тех, что приведены в книге. Наши коллеги, обучающие алгоритмам, распределенным системам, искусственному интеллекту и другим разделам информатики, должны иметь возможность при желании использовать ПО для решения собственных задач.
- Выбранная тема должна требовать применения графики и средств мультимедиа, предполагать продвинутый графический интерфейс пользователя.
- В отличие от многих видеоигр, наша продукция не должна содержать жестокости и агрессии, неприемлемых для университетского образования.

Проблемная область, выбранная нами, – транспортная система города: моделирование, планирование, имитация, отображение, статистика. Библиотека Traffic – это не просто приложение, выполняющее определенную работу, это библиотека классов, обеспечивающая повторно используемые компоненты, из которых студенты и преподаватели могут строить

приложения. Она содержит элементы ГИС (Географической Информационной Системы) в более скромном варианте, поддерживая, тем не менее, механизмы графического отображения.

Как пример, эта книга использует Париж с его улицами и транспортной системой. Так как описание города представляется отдельным файлом в формате XML, можно без труда перейти к любому другому городу с его транспортной системой. На второй неделе обучения несколько студентов ETH спонтанно подготовили файл, представляющий транспортную систему Цюриха, который с тех пор и используется.

Самое первое приложение, которое строят студенты, содержит 12 строчек. При его выполнении отображается карта, подсвечивается Парижское метро на карте, выбирается предопределенный маршрут и визуализируется турист, следующий маршрутом, в стиле видеоигры с анимацией. Вот код этого приложения:

```
class PREVIEW inherit
    TOURISM
feature
    explore
            -- Показать город и маршрут.
        do
            Paris.display
            Louvre.spotlight
            Metro.highlight
            Route1.animate
        end
end
```

Алгоритм включает только четыре оператора, выполняемых при работе программы, и все же его эффект выразителен благодаря механизмам Traffic.

Несмотря на доверие к содержанию существующего ПО, я стараюсь избежать ощущения «магии». Фактически все можно объяснить на подходящем уровне абстракции. Мы никогда не говорим: «*делайте все, как я сказал. Поймете все позже, когда подрастете*». Такое отношение не годится ни при обучении студентов, ни при воспитании собственных детей. В нашем примере даже предложение наследования **inherit** может быть объяснено простым способом. Я, конечно, не излагаю теорию наследования, но говорю студентам, что класс *TOURISM* является классом-помощником (helper class), в котором вводятся предопределенные объекты, такие, как *Paris*, *Louvre*, *Metro* и *Route1*, и что новый класс может «наследовать» от такого существующего класса, получая доступ к его компонентам (features). Говорится также, что нет необходимости рассматривать детали класса *TOURISM*, но это можно сделать, если чувствуете в себе рождение «инженерного чувства», стремление выяснить, как это все устроено.

Правило, позволяющее студентам постепенно постигать суть, всегда абстрактно и обходится без обмана.

От программирования к инженерии программ

Программирование – сердце инженерии программ, но лишь часть инженерии. Концепции инженерии направлены на разработку систем, которые могут быть большими, разрабатываться в течение долгого периода, подвержены изменениям, удовлетворяют строгим критериям качества, временных сроков и стоимости. Новичков обычно этому не учат, но важно

обеспечить, по меньшей мере, введение в предмет, которое у нас появляется в последней главе. Сюда включается анализ требований, поскольку программисты, которых мы учим, должны быть не только «технарями», но и уметь общаться с потребителями и понимать их потребности. Обсуждаются грани качества ПО, введение в модели жизненного цикла, концепции «быстрой» (agile) разработки, технологии страхования качества, Модели Потенциальной Зрелости (Capability Maturity Models).

Этот обзор дополняется рассмотрением инструментария, включающего компиляторы, интерпретаторы и систему управления конфигурацией.

Терминология

Ясность мышления подразумевает ясность в использовании слов. Я уделяю особое внимание согласованной и точно определенной терминологии. Наиболее важные определения понятий появляются в рамках.

Каждая глава заканчивается разделом «Новый словарь», в котором перечисляются все введенные термины. Первое из упражнений для студентов состоит в том, что они должны сформулировать точные определения каждого термина. Это дает возможность протестировать понимание идей, введенных в главе.

Выборы технологий

Эта книга основана на комбинации технологий: ОО-подхода, Проектирования по Контракту, Eiffel как языка проектирования и программирования. Важно подтвердить справедливость этих выборов и объяснить, почему не был выбран другой язык, например, язык Java, один из современных языков программирования.

Объектная технология

Большинство вводных курсов теперь используют ОО-язык, но не всегда следуют ОО-путем. Немногие решаются применять чисто объектное мышление в элементарной части курса. Слишком часто первые программы строятся на статических функциях (вызов функций в языках C++ и Java не требует целевого объекта). Создается впечатление, что прежде чем студенты окунутся в глубины современных технологий, они должны пройти весь сложный путь, который проходили их учителя. Этот подход предпочитает порядок «снизу вверх», и классы и объекты даются тем, кто сумел подняться на «Парнас» — вершину классических программистских конструкций.

Нет настоящих причин так относиться к ОО-подходу. Он позволяет строить программную систему как ясную и естественную *модель* концепций и объектов, с которыми он имеет дело. Если он так хорош, то он должен быть хорош для всякого, в том числе и для новичков. Следуя совету знакомого официанта в Санта-Барбаре, чей кофе сыграл свою зажигательную роль в написании этой книги: «*Жизнь полна неожиданностей — ешьте вначале десерт!*»

Классы и объекты появляются с самого начала и служат основой всей книги. Мы обнаружили, что новички с энтузиазмом воспринимают объектную технологию, если концепции введены без всяких оговорок, и воспринимают ее как правильный современный способ программирования.

Одно из принципиальных следствий объектной технологии состоит в том, что естественно вводится понятие *модели*. Появление «архитектуры, управляемой моделью» (model-driven architecture) отражает осознание идеи, центральной для ОО-технологии: успешная разработка программной системы базируется на конструировании моделей физических и концепту-

альных систем. Классы, объекты, наследование и связанные методики дают прекрасную основу для обучения эффективным методикам моделирования.

Объектная технология не подразумевает исключения традиционных подходов. Скорее, она относит традиционные механизмы к специальным случаям: ОО-программа сделана из классов, при выполнении она оперирует с объектами. Но классы содержат методы (процедуры и функции), а объекты содержат поля, с которыми программа оперирует как с обычными переменными. Как статическая структура программ, так и динамическая структура выполнения покрывает традиционные концепции. Мы абсолютно уверены, что студенты должны владеть традиционным приемами: алгоритмическими доказательствами, переменными и присваиванием, управляющими структурами, указателями (у нас рассматривается непростая задача обращения связного списка, что редко бывает в традиционных вводных курсах), процедурами и рекурсией. Студенты должны уметь строить программу с нуля.

Eiffel и Проектирование по Контракту

Мы основываемся на Eiffel и среде разработки EiffelStudio, которую студенты могут загружать из сайта www.eiffel.com. Университеты могут также инсталлировать эту свободно распространяемую версию (и при желании получать поддержку). Этот выбор непосредственно поддерживает педагогические концепции этой книги:

- язык Eiffel объектно ориентирован без всяких компромиссов;
- он обеспечивает хорошую основу для изучения других языков программирования, таких как Java, C#, C++ и Smalltalk (в приложениях, примерно на тридцати страницах каждое, дается сравнительное описание основ трех из этих языков);
- Eiffel прост для начального обучения. Концепции могут вводиться постепенно, без учета влияния тех элементов, что еще не изучены;
- среда разработки EiffelStudio использует современный, интуитивный графический интерфейс – GUI. Ее свойства включают изощренные средства просмотра, редактирование, отладчик с уникальной возможностью откатов, автоматическое построение документации (HTML или другие форматы), программные метрики, современные механизмы автоматического тестирования. Она позволяет по коду автоматически строить диаграммы, отражающие архитектуру проекта. Более того, она позволяет пользователю нарисовать диаграмму, по которой будет создан код, и процесс этот может работать в циклическом режиме;
- EiffelStudio доступна на многих платформах, включая Windows, Linux, Solaris и Microsoft.NET;
- EiffelStudio включает множество тщательно написанных библиотек, поддерживающих повторное использование и являющихся основой для библиотеки Traffic. Среди них упомянем *EiffelBase*, реализующую фундаментальные структуры и поддерживающую изучение алгоритмов и структур данных в части III, *EiffelTime* для работы с датами и временем, *EiffelVision* для переносимой графики и *EiffelMedia* для анимации и мультимедиа;
- в отличие от средств, спроектированных исключительно для образовательных целей, Eiifel коммерчески используется для критически важных приложений. Язык и среда разработки применяются в банковских системах с инвестициями в сотни миллиардов долларов, в системе здравоохранения, для сложного моделирования. По моему мнению, это является основой для эффективного обучения программированию: язык, по-настоящему хороший для профессионалов, должен быть также хорош и для новичков;
- приняты международные стандарты языка Eiffel – ISO и ECMA. Для преподавателей крайне важно, чтобы язык программирования, поддерживающий курс, был стандарти-

зован, особенно по стандарту ISO (International Standards Organization), поскольку это дает гарантию стабильности и четких определений;

Текст стандарта ISO можно найти на tinyurl.com/y5abdx. Аналогичный текст в версии ECMA находится на tinyurl.com/cq8gw.

- Eiffel — не просто язык программирования. Это Метод, чья первичная цель – помимо записи алгоритмов для компьютера – поддерживать рассуждения о задачах и способах их решения. Он позволяет учить бесшовному подходу, расширяемому на весь жизненный цикл создания ПО, от этапов анализа и проектирования до программирования и сопровождения. Эта концепция бесшовной разработки, поддерживается циклическим средством построения диаграмм Diagram Tool среды разработки EiffelStudio и согласована с преимуществами моделирования, предоставляемыми объектной технологией.

Для поддержки этих целей Eiffel непосредственно реализует концепцию **Проектирования по Контракту,** разработанную совместно с Eiffel и тесно связанную как с методом, так и с языком. Поставляя классы с предусловиями, постусловиями, инвариантами классов, мы позволяем студентам использовать более систематический подход, чем это обычно делается, и готовить их тем самым к успешной профессиональной разработке, которая способна создавать системы, не содержащие «багов».

Не следует недооценивать роль синтаксиса как для начинающих, так и для опытных пользователей. Синтаксис языка Eiffel — иллюстрированный коротким примером программы (класс Preview) — облегчает обучение, повышает читабельность программ, помогает сражаться с ошибками.

- Язык избегает криптографических символов.
- Каждое резервированное слово является полным словом английского языка без использования аббревиатур (*INTEGER*, а не int).
- Знак равенства = имеет тот же смысл, что и в математике, не нарушая устоявшейся годами математической традиции.
- Точки с запятой не являются абсолютно необходимыми. В большинстве современных языков программные тексты наперчены точками с запятыми, завершающими объявления и операторы языка. Большей частью оснований для подобной разметки текста нет. Требуемые в некоторых местах, они не являются необходимыми в других случаях. Причины их появления непонятны начинающим и могут служить источником ошибок. В Eiffel точка с запятой возможна как разделитель, на любом уровне программы. Это ведет к аккуратному внешнему виду программы, как можно видеть на любом примере программы этой книги.

Поощрение аккуратности в программных текстах должно быть одной из педагогических целей. Eiffel включает точные правила стиля, объясняемые на всем протяжении курса, показывая студентам, что хорошее программирование требует внимания как к высокоуровневым концепциям, так и к низкоуровневым деталям синтаксиса и стиля — качество в большом и качество в малом.

Подводя итоги: хороший язык программирования должен давать пользователям возможность сосредоточиваться на концепциях, а не на нотации. В этом цель использования Eiffel для обучения: студенты должны думать о своих задачах, а не об Eiffel.

Почему не Java?

Так как в курсах в настоящее время часто применяется Java или C#, следует пояснить, почему мы не следуем такой практике. Язык Java, наряду с языками C#, C++ и C, следует знать, но он не подходит на роль первого языка обучения. Слишком большой багаж знаний требу-

ется накопить, прежде чем студенты смогут думать о своих задачах. Это можно видеть на примере «Hello World» — первой программы на Java:

```
class First {
    public static void main(String args[])
    { System.out.println("Hello World!"); } }
```

Появляются концепции, каждая из которых мешает обучению. Почему «public», «static», «void»? (Конечно, я могу сделать мою программу общедоступной – *public*, если вы настаиваете, но как понять, что в результате моих усилий возникнет пустота – *void*?) Эти ключевые слова не имеют ничего общего с целью моей программы, и студенты начнут понимать их смысл, по меньшей мере, через несколько месяцев, но должны включать их в свои тексты как магические заклинания, чтобы их программы работали. Для преподавателей это означает, что они должны давать некоторые конструкции без понимания их смысла. Как отмечалось ранее, стиль «*Вы поймете, когда подрастете*» — не лучший из педагогических приемов. Eiffel защищает нас от этого: мы можем объяснить каждую используемую конструкцию языка при первом ее появлении.

ОО-природа языка и его простота играют роль. Есть некоторая ирония в том, что каждая Java-программа, начиная с простейшего примера, как показано выше, использует как главную функцию статическую функцию (static), что нарушает принципы ОО-стиля программирования. Конечно, есть люди, которым не нравится идея использовать ОО-стиль для начального курса. Но уж если вы выбрали работу с объектами, будьте добры быть последовательными. В какой-то момент студенты поймут, что фундаментальная схема – та, которую, как вы говорили, следует использовать, — вовсе не является ОО-схемой. С каким лицом вы будете отвечать на этот неизбежный вопрос?

Синтаксис, как отмечалось, имеет значение. В первом примере на Java студент должен управлять странными скоплениями символов, подобно финальным «»); } }». Они приводят глаз в замешательство, и роль их не очевидна. В этом скоплении точный порядок символов важен, но его трудно объяснить и запомнить. Почему следует ставить точку с запятой между закрывающими скобками – круглой и фигурной? Есть ли пробел после точки с запятой, а если да, то нужен ли он? Вместо того чтобы сконцентрироваться на концепциях программирования, студент должен обнаруживать тривиальные ошибки, приводящие к загадочным для него результатам.

Еще один постоянный источник недоразумений – это использование знака равенства «=» для присваивания, наследованного от языка Фортран через C. Как много студентов, начинающих обучение с Java, удивляются, почему имеет смысл a = a + 1 и, как отмечал Вирт [15], почему a = b не то же самое, что и b = a?

Несогласованности создают трудности. Почему, наряду с полными словами подобно «static», используются аббревиатуры, такие как «args» и «println»? Студенты из первого знакомства с программированием извлекают урок, что согласованность не требуется, что количество нажатий на клавиши может быть важнее ясности имен (в базисной библиотеке Eiffel операция перехода на новую строку называется *put_new_line*). Если позднее мы введем методологическое правило, требующее от студентов выбирать ясные и согласованные имена, то едва ли они будут воспринимать нас со всей серьезностью. «*Делай, как я говорю, а не так, как я делаю*» — сомнительный педагогический принцип.

Приведу еще один пример. При описании (глава 17) необходимости механизма, рассматривающего операции как объекты, подобно агентам Eiffel или замыканиям в других языках, мы должны были объяснить, как справляются с этой проблемой в языках типа Java, где та-

кие механизмы отсутствуют. Так как мы использовали итераторы в качестве мотивационного примера, мы были счастливы обнаружить, что на странице Sun, описывающей «внутренние классы» Java, приведен код для проектирования итератора, который можно было бы прекрасно использовать в качестве модели.

Смотри tinyurl.com/c4oprq (архив java.sun. com/docs/books/tuto-rial/java/javaOO/innerclasses.html, Oct. 2007; теперь страница использует другой пример).

Но тогда он включал следующее объявление:

```
public StepThrough stepThrough() {
            return new StepThrough();
        }
```

Вероятно, я смог бы объяснить его закаленным трудностями программистам. Но нет способа, позволяющего мне объяснить его начинающим студентам, — и я восхищаюсь тем, кто смог бы сделать это. Почему StepTrough появляется три раза? Означает ли это каждый раз одно и то же? Является ли изменение буквенного регистра (StepThrough vs stepThrough) значимым? Вообще, что это все может означать? Очень быстро вводный курс программирования превращается в болезненное толкование языка программирования, оставляя немного времени для настоящих концепций. По словам Алана Перлиса:

> «Язык программирования является низкоуровневым, если он требует внимания к незначимым вещам».

Эпиграмма #8, доступная на www-pu.informa-tik.uni-tuebin-gen.de/users/klaeren/ epigrams.html.

Свой вклад в трудности использования языка Java в начальном курсе вносит та свобода, с которой язык оперирует с ОО-принципами. Например:

- если x обозначает объект и a – один из атрибутов соответствующего класса, то по умолчанию разрешается писать $x.a = v$, чтобы присвоить новое значение полю a объекта. Это нарушает сокрытие информации и другие принципы проектирования. Для управления этим необходимо скрывать атрибут, дополняя его специальными модификаторами. Для преподавателя в такой ситуации возникает выбор: либо заставлять студентов на первых шагах добавлять текст, носящий для них характер шума, либо разрешать на первых порах плохой стиль проектирования, а потом с трудом их переучивать;
- Java строго разделяет полностью абстрактные модули, называемые интерфейсами, от полностью реализованных – классов. Одно из преимуществ механизма классов, введенное еще в Simula 67, состоит в существовании полного спектра возможностей между этими двумя полюсами. Эта идея является центральной в обучении ОО-методу, в частности, обучению проектированию. Можно начинать определение некоторого понятия с полностью отложенного (deferred) абстрактного класса, затем вы постепенно уточняете его, используя наследование, приходя к полностью эффективному классу. Классы на промежуточном уровне в этом процессе частично отложены, частично эффективны. Java не позволяет использовать такой подход. Если вам нужно скомбинировать несколько абстракций, все они, за исключением максимум одной, должны быть интерфейсами.

Можно привести еще много примеров подобного влияния языка Java на процесс обучения. Типичную реакцию при переходе на Eiffel выразил один из программистов в своем письме: «*Я много писал на C++ и Java и тратил уйму усилий на изучение груза скучной компьютерной чепухи. С Eiffel не замечаешь программирования, и я трачу свое время на размышления о задаче*».

Причина, по которой во вводном курсе часто используется C++ и Java, состоит в том, что рынок требует программистов, работающих на этих языках. Это разумный аргумент, но он применим к учебному плану в целом, а не к первому курсу. Программирование на уровне, требуемом современными стандартами, достаточно сложно, а потому следует использовать наилучшие обучающие инструменты. Если бы рынок диктовал требования к обучению, мы бы никогда ранее не использовали Паскаль (многие годы служивший языком начального обучения), не говоря о языке схем. Если при обучении следовать тенденциям, существующим в программистском мире, то мы бы проходили периоды Fortran, Cobol, PL/I, Visual Basic, C, и программисты, закончившие обучение, обнаруживали бы, что их знания устарели, при каждом повороте великого колеса моды – через несколько лет. Наша задача – выпускать тех, кто решает задачи и может быстро адаптироваться к эволюциям нашей дисциплины.

Не следует позволять кратковременным требованиям рынка управлять принципами обучения. Скажем так: если вы считаете C++ или Java идеальными обучающими средствами, используйте их. Вероятно, в этом случае данная книга вам совсем не будет по душе. Но если вы согласны с ее подходом, не позволяйте себя запугать высказываниями некоторых студентов или их родителей, что вы исповедуете «академический» подход. Объясните им, что вы используете лучший подход из того, что вы знаете, что тот, кто поймет суть программирования, получит эти навыки на всю жизнь, и что любой неплохой инженер ПО может быстро освоить новый язык за завтраком, даже если он не прошел его в других курсах учебного плана. Что же касается характеристики «академический подход», то сошлитесь на сайт eiffel.com, где приведен впечатляющий список критически важных коммерческих приложений, реализованных на Eiffel в различных компаниях, часто в ситуациях, когда их попытки реализации на других языках потерпели неудачу.

Языки Java, C#, C++ и C являются на ближайшие несколько лет багажом инженера ПО, они важны, что и нашло отражение в четырех приложениях к данной книге. Эта цель, однако, никак не связана с теми методиками, которые следует использовать в вводном курсе.

По нашим опросам [13], примерно 50% студентов уже использовали Java или C++ до изучения вводного курса.

В какой-то момент студенты познакомятся с этими языками; в наши дни редкий учебный план не знакомит, по крайней мере, с одним из них. Но, насколько я знаю, ни один из вводных курсов не рассказывает обо всех средствах, так что придется все равно изучать разные языки вне зависимости от начального языка обучения.

Языки программирования и культура программирования, связанная с каждым из них, сами по себе являются интересным объектом изучения. Наша группа в ETH, которая учит вводному курсу на Eiffel, имеет на старших курсах опыт преподавания специальных языков: «В глубины Java», «В глубины C#». Когда вы понимаете концепции программирования, вы готовы к овладению различными языками, изучение Eiffel и его объектной модели помогает вам стать лучшим C++ или Java-программистом.

> Я работаю одновременно как в индустрии, так и в академической сфере, и мне ежемесячно приходится читать десятки резюме. Все их обладатели гордятся одними и теми же навыками, включающими опыт работы на C++ и Java. Это уже никого не поражает и не позволяет выделить претендента из толпы ему подобных. Действительным преимуществом может служить знание ОО-подхода и его применения в ПИ, о чем мог бы свидетельствовать учебный план по Eiffel и Проектированию по Контракту. Вполне можно представить учебный план, основанный на C++ и не содержащий настоящего понимания ОО-концепций; с Eiffel это менее вероятно. Компетентные ра-

> ботодатели понимают, что, помимо непосредственных навыков, важна глубина понимания проблем ПО и способность к профессиональной разработке в течение длительного времени. Все усилия, предпринятые в этой книге, и использование Eiffel направлены на эти цели.

Здесь стоит еще раз процитировать Алана Перлиса: *«Язык, который не воздействует на способ вашего размышления о программировании, не стоит изучения»*.
Эпиграмма #19

Насколько формально?

Подход «Проектирование по Контракту» позволяет в «мягкой дозе» познакомить студентов с «формальными» методами разработки ПО.

Формальные методы необходимы при разработке ПО. Любой серьезный учебный план должен включать, по крайней мере, один курс, полностью посвященный математическим основам разработки ПО, математическому языку задания спецификаций. Эти идеи должны оказывать влияние на весь учебный план, хотя, как обсуждалось ранее, нежелательно использовать для начинающих полностью формальный подход. Проблема в том, чтобы практические навыки и формальные методы представить как дополняющие друг друга аспекты, тесно связанные и в равной степени обязательные. Подход «Проектирование по Контракту» позволяет решить эту задачу.

Начиная с первых примеров спецификации интерфейсов, методы обладают предусловиями и постусловиями, а классы – инвариантами. Как и должно, эти концепции обсуждаются и вводятся в походящем контексте. Следует отметить, что многие программисты все еще боятся их, и большинство языков программирования не поддерживает контракты как естественный способ рассуждения о программах. Не пугая студентов тяжело воспринимаемыми формальными методами, мы открываем простой способ их введения, который полностью будет оценен по мере приобретения соответствующего опыта программирования.

Введение формализации может хорошо сочетаться с практическим подходом. Например, при изучении циклов они рассматриваются как механизм аппроксимации, вычисляющий решение на расширяющемся подмножестве данных. В этом случае понятие инварианта цикла становится естественным, как ключевое свойство поддержания аппроксимации на каждом шаге цикла.

Эта ставка на практичность отличает «Проектирование по Контракту» от полностью формальных методов, используемых в некоторых вводных курсах, в которых преподаватели исходят из того, что вводный курс по программированию является математическим курсом. Иногда дело доходит до того, что студенты не работают с компьютером в течение семестра или всего учебного года. Есть риск, что вместо желаемого эффекта будет достигнут противоположный результат.

Студенты, в частности, те, кто программировал ранее, осознают, что они могут создавать программы — несовершенные, но работающие — без всякого тяжелого математического аппарата. Если вы скажете им, что это невозможно, то можете просто потерять контакт с ними, и в результате они могут просто отвергать формальные методы как неподходящие, не желая пользоваться как простыми идеями, которые могли бы им помочь непосредственно сейчас, так и более продвинутыми, полезными позже. Лесли Лемпорт (Leslie Lamport), – которого никак нельзя обвинить в недооценке формальных методов, – указывал [6]:

> [В американских университетах] математика и инженерия полностью разделены. Я знаю уважаемый американский университет, в котором студенты на первом программистском курсе должны доказывать корректность каждой маленькой программы, которую они пишут. На их втором программистском курсе они все доказательства полностью забывают и просто учатся писать программы на С. Они не пытаются применить то, что они выучили на первом курсе, к написанию реальных программ.

Наш опыт подтверждает это. Студенты первого курса, хорошо воспринимающие Проектирование по Контракту, не готовы к полностью формальному подходу. Для выработки реального осознания преимуществ необходимо ощутить сложности разработки индустриального ПО. Но, с другой стороны, не следует допускать на первом курсе полностью неформальный подход, а годами спустя неожиданно показывать, что программирование нечто большее, чем хакерство. Подходящая методика, верю я, заключается в постепенности: вводить Проектирование по Контракту с первых шагов, сопровождая утверждением, что программирование основано на математическом стиле доказательств, и позволять учащимся овладеть практикой разработки ПО на основе умеренного формального подхода. Позже в учебном плане нужно предусмотреть курсы по таким темам, как формальная разработка и семантика языков программирования. Этот цикл может повторяться, так, чтобы теория и практика сменяли и дополняли друг друга.

Такой подход помогает студентам воспринимать концепции корректности не как академические химеры, а как естественный компонент процесса конструирования ПО.

> В том же духе ведется обсуждение высокоуровневых функциональных объектов (агенты, глава 17, и их приложения к событийно управляемому программированию в главе 18). Такой подход обеспечивает возможность простого введения в лямбда-исчисление, включая карринг – математические разделы, редко включаемые в вводные курсы, но имеющие приложения на всем протяжении изучения программирования.

Другие подходы

Рассматривая учебные планы университетов, обсуждая с преподавателями, анализируя учебники, приходишь к заключению, что существуют четыре подхода к преподаванию вводного курса программирования:

1) сфокусированный на языке;
2) функциональный (в духе функционального программирования);
3) формальный;
4) структурный, стиль Паскаля или Ады.

Важно понять преимущества каждого из стилей и их ограничения.

Первый подход сегодня наиболее часто встречается. Он фокусируется на конкретном языке программирования, обычно Java или C++. Преимущество — в практичности и легко создаваемых упражнениях (есть риск использования студентами приема Google-and-Paste). Недостаток в том, что слишком много внимания уделяется выбранному языку в ущерб фундаментальным концептуальным навыкам.

Второй подход демонстрирует известный курс в МТИ (Массачусетском технологическом институте), который основан на схемном языке Sheme – функциональном языке программирования [1], устанавливающем некоторый стандарт для амбициозного учебного плана. Делаются также попытки использовать такие языки, как Haskell, ML или OCaml. Этот метод хорош для обучения навыкам логического вывода, лежащим в основе программирования. Мы стара-

емся сохранить эти преимущества, так же как и отношение к математике, используя логику и Проектирование по Контракту. Но, по моему мнению, ОО-технология дает студентам лучший способ справляться с проблемами конструирования программ. Не только потому, что ОО-подход соответствует практике современной индустрии ПО, высказывающей мало интереса к функциональному стилю программирования; более важно, что он лучше соответствует приемам построения систем и архитектуре ПО, а это является центральной задачей образования.

Хотя, как мы отмечали, учебный план не должен быть рабом доминирующих технологий, просто потому, что они доминируют, использование приемов и методик, далеких от практики, может привести нас к уже упомянутому риску потери связи со студентами, особенно продвинутыми. Алан Перлис высказал это менее дипломатично: «*Чисто функциональные языки плохо функционируют*».

Эпиграмма #108.

Я полагаю, что операционный, императивный аспект разработки ПО — это фундаментальный компонент дисциплины программирования, без которого исчезают наиболее трудные проблемы. В функциональном программировании он недооценивается и рассматривается как помеха реализации. Мне кажется, мы не особенно поможем студентам, защищая их от этих аспектов в начале образования. Им придется полагаться на собственные силы, когда они встретятся с ними позднее (добавьте к этому, что функциональное программирование в наши дни требует знакомства с монадами, — коль есть выбор, то я предпочитаю учить присваиванию, а не теории категорий).

> Стоит отметить, что ОО-программирование математически вполне респектабельно благодаря теории абстрактных типов данных, на которой оно основано, а в Eiffel – благодаря контрактам. Как и в любом другом подходе, здесь хватает интеллектуальных вызовов. Рекурсия, один из замечательных механизмов функционального программирования, широко освещается в данной книге (глава 14).

Некоторые комментарии к функциональному программированию также применимы и к третьему подходу к обучению, покоящемуся на формальных методах. Как обсуждалось выше, формальные методы на начальном этапе преждевременны. Практическим эффектом такого подхода может быть возникающая у студентов уверенность, что академическая компьютерная наука не имеет ничего общего с практикой инженерии ПО.

Четвертый широко используемый подход, пионером введения которого был ETH, корнями уходит в семидесятые годы – годы становления структурного программирования. Этот подход распространен и до сих пор. Поддерживающим языком программирования является обычно Паскаль или один из его потомков — Modula-2, Oberon или Ada. Подход нашей книги является наследником этой традиции, в котором объектная технология рассматривается как естественное расширение структурного программирования, фокусируясь на рассмотрении масштабируемого программирования, отвечающего вызовам 21-го века.

Покрытие тем

Эта книга разделена на пять частей.

Часть I вводит основы. Она определяет строительные блоки программы, от объектов и классов до интерфейсов, управляющих структур и присваивания. Особое внимание уделяется понятию контракта. Студенты учатся на абстрактном, но вместе с тем точном описании используемых ими модулей и должны применять такой же интерфейс для создаваемых модулей. В главе 5 «Немного логики» вводятся ключевые элементы пропозиционального ис-

числения и исчисления предикатов. Оба исчисления создают основу дальнейших обсуждений. Возвращаясь к программированию, в последующих главах мы рассматриваем создание и структуру объектов. В этих главах устанавливается моделирующая мощь объектов и необходимость при построении объектной модели отражения реальной структуры моделируемой внешней системы. После введения концепций структурирования программы разбирается присваивание, ссылки, ссылочное присваивание и интересные задачи, возникающие при работе со связанными списками.

Часть II, озаглавленная «Как это все работает», представляет взгляд изнутри. Она начинается с основ организации компьютера, рассматриваемых с позиций программиста и включающих только базисные концепции. Далее изучаются методы описания синтаксиса – БНФ и приложения, языки и инструментарий программирования. Две следующие главы посвящены центральным темам: синтаксису и способам его можно описания, включая БНФ и введение в теорию конечных автоматов, обзор языков программирования, инструментария и сред разработки ПО.

Часть III посвящена фундаментальным структурам данных и алгоритмическим методам. Она включает три главы, рассматривающие:

- фундаментальные структуры данных – глава не является пересказом курса «Алгоритмы и структуры данных», часто следующего за вводным курсом; в ней вводятся такие понятия, как универсальность и алгоритмическая сложность. Здесь же рассматриваются некоторые важные структуры данных – массивы, списки, хэш-таблицы;
- рекурсию, включая бинарные деревья, в частности, бинарные деревья поиска. В главе дается введение в теорию неподвижной точки, введение в методы реализации рекурсии;
- детальное исследование одного интересного семейства алгоритмов — топологической сортировки, выбранной за ее свойства, которые позволяют демонстрировать как проектирование алгоритма, так и возникающие проблемы ПИ. Обсуждение покрывает математическое обоснование алгоритма, последовательную разработку алгоритма, направленную на эффективное выполнение, и инженерию разработки интерфейса – API (Application Programming Interface), направленного на удобство практического использования.

Часть IV ведет в глубины ОО-техники. В первой главе рассматривается наследование, приводятся многие детали, редко появляющиеся во вводных курсах, такие как образец (pattern), Visitor (Посетитель), дополняющий базисный механизм наследования для случая расширения операций существующего типа. Следующая глава посвящена современной теме развития ОО-метода — функциям, рассматриваемым как объекты. Терминология не устоялась, в разных языках эти понятия называют замыканиями, делегатами, *агентами* (используемый здесь термин). Эта глава включает введение в лямбда-исчисление. Последняя глава применяет технику агентов к важному стилю программирования – программированию, управляемому событиями. Здесь появляется возможность рассмотреть еще один стандартный образец проектирования – «Observer» («Наблюдатель») и проанализировать его ограничения.

Часть V добавляет финальное измерение, выходящее за пределы простого программирования. Вводятся концепции ПИ для больших, долго живущих проектов.

Приложения, уже упомянутые, обеспечивают введение в языки программирования, с которыми студенты должны быть знакомы: Java, C#, C++ – мост между ОО-миром и языком С.

Благодарности

Некоторые элементы предисловия для преподавателей взяты из ранних публикаций: [7], [8], [9], [10], [12].

Источником для этой книги послужил, как отмечалось, читаемый в ETH курс «Введение в программирование», постоянно поддерживаемый всем окружением ETH. Особую благодарность приношу Ректорату (который помог реализовать начальную разработку библиотеки Traffic, благодаря предоставленному FILEP гранту), а также факультету информатики, в частности его главе Питеру Видмайеру (Peter Widmayer), кто первый предложил мне прочесть вводный курс и приложил много усилий для координации моего и его собственного курса.

Я вел этот курс в первом семестре, начиная с 2003 года, и признателен выдающейся команде ассистентов за эффективную организацию зачетов, консультирование студентов, продумывание упражнений и вопросов для экзамена, опросов студентов, организации студенческих проектов, создания поддерживающих документов и учебных материалов, а также за замену меня на лекциях в случаях моего отсутствия. Это позволило мне сконцентрироваться на разработке педагогических концепций и основном материале, будучи уверенным, что все остальное будет работать. Я также благодарен сотням студентов, принявших этот курс вместе с моими пробами и ошибками и обеспечивших лучшую обратную связь, о которой можно только мечтать, — великолепные проекты ПО.

Смотри, например: games.ethz.ch

Ассистентами курса в 2003-2008 годах были: Волкан Арслан, Стефания Бальцер, Тилл Бэй, Карин Безаулт, Бенно Баумгартен, Рольф Брудерер, Урсина Калуори, Роберт Карнеки, Сюзанна Превитали, Стефан Классен, Йорг Дерунгс, Илинка Кьюпа, Иво Коломбо, Адам Дарвас, Питер Фаркас, Майкл Гомец, Себастьян Грубер, Беат Херлиг, Маттиас Конрад, Филипп Крэхенбюл, Герман Лехнер, Андреас Лейтнер, Рафаэль Мак, Бенджамин Моранди, Ян Мюллер, Мари-Элен Ниеналтовски, Петер Ниеналтовски, Мишела Педрони, Марко Пиччиони, Конрад Плано, Надя Поликарпова, Маттиас Сала, Бернд Шоллер, Вольфганг Шведлер, Габор Сабо, Себастьян Ваукулер, Джи Вэй и Тобиас Видмер.

Хочется особо отметить Мануля Ориоля за его участие в наших исследованиях, Тилла Бэя (за разработку библиотеки EiffelMedia, лежащей в основе студенческих проектов, библиотеки EiffelVision, разработанной при написании диплома, проекта Origo и сайта origo.ethz.ch как части его PhD диссертации), Карину Безаулт, Илинку Кьюпа, Андреаса Лейтнера, Мишелу Педрони и Марко Пиччиони (все они старшие преподаватели и принесли неоценимую пользу во многих начинаниях). Клаудиа Гюнтхарт обеспечивала выдающуюся административную поддержку.

ПО Traffic сыграло особо важную роль в подходе, развиваемом в этой книге. Текущая версия разрабатывалась в течение нескольких лет Мишелой Педрони, начавшей с оригинальной версии, которая была написана Патриком Шёнбахом под руководством Сюзанны Пре-

витали; несколько студентов внесли свой вклад в разработку под руководством Мишелы Педрони в разных семестрах в своих магистерских работах, в частности (в хронологическом порядке): Марсель Кесслер, Рольф Брудерер, Сибилла Арегер, Валентин Вюсгольц, Стефан Даниэль, Урсина Калуори, Роджер Кюнг, Фабиан Вюст, Флориан Гельдмахер, Сюзанна Каспер, Ларс Крапф, Ганс-Герман Джонас, Майкл Кэзер, Николя Бизирианис, Адриан Хельфенштейн, Сара Хаузер, Мишель Крочи, Алан Фехр, Франциска Фритчи, Роджер Имбах, Матиас Бюхлман, Этьен Рэйхенбах и Мария Хьюсман. Их роль была неоценимой в привнесении пользовательского взгляда на продукт, учитывая, что они уже прослушали курс с ранней версией системы Traffic. Мишела Педрони занималась оркестровкой согласования ПО и книги, а также принимала участие в разработке педагогического подхода – обращенного учебного плана, подхода «извне – внутрь», поддерживающего инструментария (смотри trucstudio.origo.ethz.ch). Мари-Элен Ниеналтовски также принимала участие в наших педагогических работах, создав систему TOOTOR, помогающую студентам осваивать материал; она попыталась привнести наш подход в Биркбек колледж университета Лондона (Birkbeck College, University of London).

Я благодарен моим коллегам по факультету Computer Science Department (Departement Informatik) в ETH за вдохновляющие дискуссии попроблемам обучения; я должен поблагодарить за критику и предложения Уолтера Гандера (кто помог мне улучшить важные примеры), Густаво Алонсо, Уэля Маурера, Юрга Гюткнехта, Тома Гросса, Питера Мюллера и Питера Видмайера.

Вне ETH я извлек много пользы от общения с преподавателями, включая Кристину Мингинс, Джонатана Остроффа, Джона Поттера, Рихарда Патисса, Джин-Марка Джезекуэля, Владимира Биллига, Анатолия Шалыто, Андрея Терехова и Юдифь Бишоп.

Для всех моих публикаций за последние годы, включая и эту книгу, огромную ценность представляет выдающаяся работа по созданию библиотек и среды разработки EiffelStudio, созданной в фирме Eiffel Software всей командой разработчиков и ведущей ролью Эммануэля Стапфа.

Я также благодарен комитету по стандартам ECMA International TC49-TG4, взявшему на себя заботы о стандарте ISO Eiffel, принимавшему участие во всех рассмотрениях по улучшению и расширению языка проектирования и учету потребностей начинающих студентов. Свой вклад внесли Эммануэль Стапф, Марк Ховард, Эрик Безаулт, Ким Уолден, Зоран Симик, Пауль-Георг Крисмер, Роджер Осмонд, Пауль Кохен, Кристина Мингинс и Доминик Колнет.

Обсуждения на форуме Eiffel Software: groups.eiffel.com также были полезны.

Перечисление даже подмножества людей, оказавших на меня влияние, заняло бы несколько страниц. Многие из них цитируются в тексте, но один — нет: в теме «представление рекурсии» заимствованы некоторые из идей онлайновых записей лекций Андриеса ван Дама из университета Брауна.

Многие люди представили свои комментарии к черновым вариантам книги. Я должен отметить в частности Bernie Cohen (хотя его принципиальное влияние начало ощущаться задолго до написания книги, когда он предложил концепцию обращенного учебного плана), Филиппа Кордела, Эрика Безаулта, Огниана Пишева и Мухаммеда Абд-Эль-Разика, также как студентов и ассистентов ETH: Карин Безаулт, Йорга Дерунгса, Вернера Диетла, Мориса Диетше, Лючин Доблис, Марка Эгга, Оливера Джегера, Эрнста Лейзи, Ханну Рёст, Рафаэля Швейцера и Элиаса Юсефи. Герман Лехнер предложил несколько упражнений. Трюгве Реинскауг внес важные комментарии в главу по событийно управляемому программированию. Я особо благодарен за внимательное чтение и поиск ошибок в последней редакции, выполненных Марко Пиччиони и Стефану ван Стадену.

Особая благодарность создателям материалов, вошедших в приложения по отдельным языкам программирования: Марко Пиччиони (Java, приложение A), Бенджамину Моранди (C#, приложение B) и Наде Поликарповой (C++, приложение C). Конечно, на мне лежит вся ответственность за любые дефекты в окончательном варианте их представления.

Не могу найти нужных слов, чтобы описать всю ценность чрезвычайно упорной и профессиональной работы по чтению финальной версии Анни Мейер и Рафаэлю Мейеру, приведшую к сотням (фактически тысячам) коррекций и улучшений.

Так много людей помогло мне, что я боюсь, что кого-то пропустил, Поэтому оставляю список открытым.

Смотри touch.ethz.ch/acknowledgmentsonline,

Я хотел бы закончить благодарностью за помощь и советы Монике Рипл из издательского отдела в Лейпциге за помощь и советы в подготовке книги к изданию, Герману Ингессеру и Дороти Глаунзингер из издательства Шпрингер за теплую и эффективную поддержку в процессе публикации.

БМ

Санта-Барбара / Цюрих, Апрель 2009

Литература

[1] Harold Abelson and Gerald Sussman: *Structure and Interpretation of Computer Programs*, 2nd edition, MIT Press, 1996.

[2] Bernard Cohen: *The Inverted Curriculum*, Report, National Economic Development Council, London, 1991.

[3] Mark Guzdial and Elliot Soloway: *Teaching the Nintendo Generation to Program*, in *Communications of the ACM*, vol. 45, no. 4, April 2002, pages 17-21.

[4] Joint Task Force on Computing Curricula: *Computing curricula 2001* (final report). December 2001, tinyurl.com/d4uand.

[5] Joint Task Force for Computing Curricula 2005: *Computing Curricula 2005*, 30 September 2005, www.acm.org/education/curric_vols/CC2005-March06Final.pdf.

[6] Leslie Lamport: *The Future of Computing*: *Logic or Biology*; text of a talk given at Christian Albrechts University, Kiel on 11 July 2003, research.microsoft.com/users/lamport/pubs/future-of-computing.pdf.

[7] Bertrand Meyer: *Towards an Object-Oriented Curriculum*, in *Journal of Object-Oriented Programming*, vol. 6, no. 2, May 1993, pages 76-81. Revised version in *TOOLS 11* (*Technology of Object-Oriented Languages and Systems*), eds. R. Ege, M. Singh and B. Meyer, Prentice Hall, Englewood Cliffs (N.J.), 1993, pages 585-594.

[8] Bertrand Meyer: *Object-Oriented Software Construction*, *2nd edition*, Prentice Hall, 1997, especially chapter 29, "*Teaching the Method*".

[9] Bertrand Meyer: *Software Engineering in the Academy*, in Computer (IEEE), vol. 34, no. 5, May 2001, pages 28-35,se.ethz.ch/~meyer/publications/computer/academy.pdf.

[10] Bertrand Meyer: *The Outside-In Method of Teaching Introductory Programming*, in Manfred Broy and Alexandre V. Zamulin, eds., Ershov Memorial Conference, volume 2890 of Lecture Notes in Computer Science, pages 66-78. Springer, 2003.

[11] Christine Mingins, Jan Miller, Martin Dick, Margot Postema: *How We Teach Software Engineering*, in *Journal of Object-Oriented Programming* (JOOP), vol. 11, no. 9, 1999, pages 64-66 and 74.

[12] Michela Pedroni and Bertrand Meyer: *The Inverted Curriculum in Practice*, in *Proceedings of SIGCSE 2006* (Houston, 1-5 March 2006), ACM, se.ethz.ch/~meyer/publications/teaching/sigcse2006.pdf.

[13] Michela Pedroni, Manuel Oriol and Bertrand Meyer: *What do Beginning CS students know?*, submitted for publication, 2009.

[14] Raymond Lister: *After the Gold Rush: Toward Sustainable Scholarship in Computing*, in *Proceedings of Tenth Australasian ComputingEducation Conference (ACE2008)*, Wollongong, January 2008), crpit.com/confpapers/CRPITV78Lister.pdf.

[15] Niklaus Wirth: *Computer Science Education*: *The Road Not Taken*, opening address at ITiCSE conference, Aarhus, Denmark, June 2002, www.inr.ac.ru/~info21/texts/2002-06-Aarhus/en.htm.

Web-адреса приходят и уходят. Все ссылки, появляющиеся в этом списке литературы и остальной части книги, были действующими на дату: April 19, 2009.

Заметки для преподавателей: что читать студентам?

Для обеспечения гибкости при чтении курса книга содержит больше материала, чем предусмотрено в обычном семестровом курсе. Следующие заметки отражают мою точку зрения на то, что следует рассмотреть в первоочередном порядке, а какие материалы могут считаться дополнительными. Эти рекомендации основаны на моем опыте и, естественно, могут быть адаптированы с учетом специфики читаемого курса и вкусов преподавателя.

- Главы 1-4, вероятно, следует рассмотреть в полном объеме, поскольку они вводят фундаментальные концепции.
- Глава 5 вводит фундаментальные концепции логики. Если студентам читается курс логики, то материал может быть дан обзорно, фокусируясь на нотации и соглашениях, принятых в компьютерной области. Я нахожу полезным рассматривать подробнее свойства импликации, изначально контр-интуитивные для студентов, а также полезно обсудить полуограниченные булевские операции (5.3), не рассматриваемые в обычной логике.
- Глава 6 по созданию объектов необходима.
- В главе 7 параграфы вплоть до 7.6, посвященные управляющим структурам, также необходимы. Оставшиеся разделы представляют детали низкоуровневых структур ветвления и некоторые языковые вариации. Следует ознакомить со структурным программированием (7.8).
- Глава 8 по процедурам и функциям, с моей точки зрения, должна быть включена полностью; в частности, полезно дать простое доказательство неразрешимости проблемы останова.
- В главе 9 разделы вплоть до 9.5 покрывают фундаментальные концепции. В 9.6 обсуждаются трудности программирования со ссылками на рассмотрение примера обращения списка, важного, но более трудного. Последний раздел по динамическим псевдонимам может быть опущен.
- Насколько подробно излагать материал главы 10, зависит от того, слушают ли студенты курс по архитектуре компьютеров. Материал главы дает основы, необходимые для понимания программирования, со ссылками.
- Глава 11, по синтаксису, содержит важный материал, но не являющийся необходимым для понимания оставшейся части книги. Полагаю полезным рассмотреть материал вплоть до раздела 11.4 (студентам необходимо понимание абстрактного синтаксиса). Если большинство студентов не будут далее слушать курс по языкам и компиляторам, то они смогут ознакомиться с базисными концепциями в последующих разделах главы.
- Глава 12 по языкам программирования и инструментарию является фоновой. Я не даю ее в лекциях, оставляя для самостоятельного знакомства.

- Глава 13 вводит фундаментальные концепции структур данных, универсальности, статической типизации, алгоритмической сложности. Разделы 13.8 (варианты списка) и 13.13 (итерации) можно опустить.
- Глава 14 посвящена рассмотрению рекурсии — более глубокому, чем это обычно делается. Я чувствую полезность удаления загадочности рекурсивных алгоритмов; необходимо показать важность рекурсии *вне* связи с алгоритмами: рекурсивные определения, рекурсивные структуры данных, рекурсивные правила синтаксиса, рекурсивные доказательства. Базисный материал расположен в начале главы 14.1–14.4, он включает обсуждение бинарных деревьев. Другие разделы могут рассматриваться как дополнительные. Алгоритмы перебора с возвратами и альфа-бета-алгоритм (14.5) являются полезными иллюстрациями приложения рекурсии. Если курс ориентирован на реализацию, то стоит рассмотреть параграф 14.9 – реализацию рекурсии. Если вы понимаете важность контрактов, то рассмотрите раздел 14.8 – рекурсия и контракты.
- В главе 15 детально обсуждается важное приложение – топологическая сортировка. Она не вводит никаких новых конструкций, так что может быть опущена или заменена другим примером. Я рассматриваю ее как пример, демонстрирующий весь процесс разработки, начиная с математических основ, переходя к алгоритму, к выбору оптимальных структур данных, к подходящей инженерии с построением API.
- В главе 16, посвященной наследованию, основными являются разделы 16.1–16.7, 16.9. Также полезно разобрать роль универсализации в 16.12. Заключительные разделы, в частности 16.14, посвященный образцу Visitor, приводят более продвинутый материал, который в большинстве курсов не излагается из-за нехватки времени. Его можно рассматривать как подготовку к последующим курсам.
- Глава 17 по агентам (замыканиям, делегатам) также вне обычного вводного курса. Но эта тема так важна для современного программирования, что, по моему мнению, должна быть рассмотрена, по крайней мере, до параграфа 17.4 включительно с примерами по численному программированию и итерацией. Обычно у меня не хватает времени на разбор дальнейшего материала, включающего мягкое введение в лямбда-исчисление, но этот материал может быть интересен для самостоятельной работы студентов с математическими склонностями.
- Если рассматривать агенты, то целесообразно в главе 18 пожинать плоды и показать те преимущества, которые дают агенты для организации механизма событий и программирования, управляемого событиями, в особенности проектирования GUI (Graphic User Interface), интересующего многих студентов. Появляется хорошая возможность изучить важный образец Observer. В нашем курсе эта и предыдущая глава рассматриваются на четырех 45-и минутных лекциях.
- Глава 19 (введение в ПИ) не является критической для вводного курса, и у меня не хватает времени рассмотреть ее (но позже в учебном плане предусмотрены курсы по архитектуре ПО и курс по ПИ). Эти темы необходимы аудитории, нацеленной на разработку индустриального, качественного ПО.
- Приложения являются сопровождающим материалом, и я не говорю о них в лекциях, хотя некоторые преподаватели посвящают некоторое время таким языкам, как Java или C++ (мы делаем это в специальных курсах, посвященных тому или иному языку программирования).

Финальные замечания: в то время как книга и курс разрабатывались параллельно, я всегда старался ввести некоторую спонтанность и посвятить часть лекций темам, *не рассматриваемым* в книге. Мне нравилось, например, изложить алгоритм, задающий расстояние между строками — *Levenshtein distance*, поскольку он дает великолепный пример полезности ин-

вариантов цикла. Без них алгоритм кажется магией, с их введением – он становится прозрачным. Некоторые из таких материалов доступны на сайте touch.ethz.ch.

Следуя этому стилю чтения лекций и полагая, что материал достаточно подробно изложен в учебнике, я применял методику Сократа, предлагая студентам предварительно изучить новый материал, превращая лекцию в ответы на вопросы студентов. Возможно, кто-то из преподавателей захочет последовать этому примеру.

Часть I

Основы

В этой вводной части мы начнем наше путешествие в мир программирования с самых его основ: объектов, классов, интерфейсов и контрактов. Мы рассмотрим поддерживающие концепции, включающие логику и внутреннее устройство компьютера, которые каждый программист должен знать.

1

Индустрия чистых идей

1.1. Их машины и наши

Инженеры проектируют и строят машины. Автомобиль – машина для путешествий, электрическая цепь – машина, преобразующая сигналы, мост – машина, позволяющая пересечь реку. Программисты – инженеры ПО – также проектируют и строят машины. Мы называем наши машины программами или системами.

Есть разница между нашими машинами и их машинами. Если вы уроните одну из их машин, то можете повредить себе ногу. Наши тоже падают, но ног не ломают.

Программы не материальны. В некотором отношении они ближе к математическим теоремам или философским высказываниям, чем к самолетам и пылесосам. И все же, в отличие от теорем и высказываний, они являются инженерными устройствами: вы можете оперировать с программами подобно тому, как вы оперируете с пылесосом, достигая определенного результата.

Так как невозможно оперировать с чистой идеей, необходимо некоторое материальное, ощутимое устройство, позволяющее оперировать с программами или, используя более общие термины, запускать или выполнять (run, execute) их. Такую поддержку осуществляет другая машина: **компьютер**. Компьютеры и связанные с ними устройства называют аппаратурой (**hardware**). Хотя компьютеры становятся все легче, они являются машинами и могут, падая, повредить ногу. Программы и все, что к ним относится, в противоположность аппаратуре называют ПО – «программное обеспечение» (**software**) — слово, появившееся в 50-е годы, когда возник интерес к программам.

Давайте посмотрим, как эти вещи работают. Вы мечтаете о машине, большой или маленькой, и описываете вашу мечту в форме программы. Программа может быть загружена в компьютер для выполнения. Сам по себе компьютер является машиной общецелевого назначения, но когда в него загружается программа, он становится специализированной машиной – материальным воплощением нематериальной машины, описанной в вашей программе.

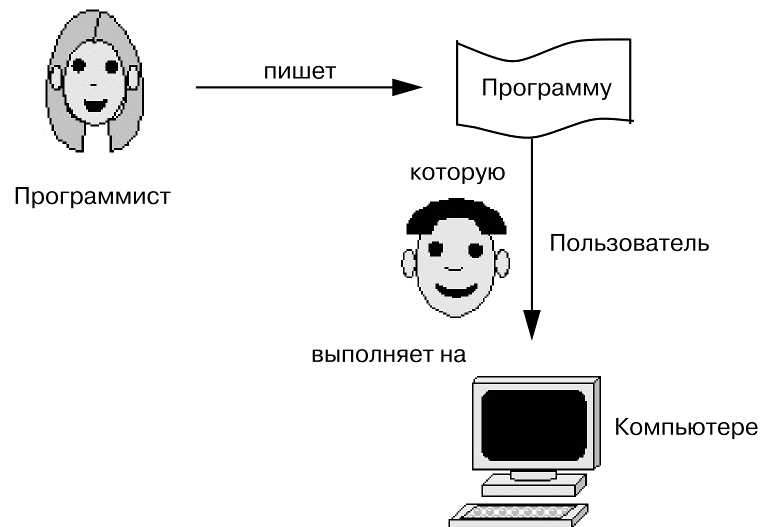


Рис. 1.1. От идеи к результатам

Пишете программу «вы», предсказуемо названные в предыдущем абзаце программистом. Другие, называемые пользователями, могут затем выполнить вашу программу на вашем или своем компьютере.

Если вам уже приходилось пользоваться компьютером, то вы запускали некоторые программы, например, бродя по Интернету или играя в игры, так что вы уже пользователь. Эта книга должна помочь вам сделать следующий шаг — стать программистом.

Циники в индустрии ПО «юзеров» презрительно называют «лузерами» (loser – проигравший). Наша цель в том, чтобы пользователи наших программ называли себя «винерами» (winner – победитель).

Нематериальная природа машин, которые мы строим, делает программирование столь привлекательным. Обладая достаточно мощным компьютером, можно определить любую машину по вашему желанию, ее операции будут требовать миллиарды и миллиарды шагов, и компьютер выполнит их для вас. Вам не понадобятся ни дерево, ни клей, ни металл, не нужен молоток, не нужно бояться, что вы что-то разобьете, подымаясь по ступенькам, или порвете одежду. Придумайте и получите. Единственное ограничение – это ваше вооображение.

На самом деле, это одно из двух ограничений; мы избегаем упоминать другое в интеллигентной компании, но, вероятно, вы сталкивались с ним уже не раз. Ограничение связано с вашей способностью совершать ошибки. Ничего личного: если вы подобны мне и многим другим людям, то вы делаете ошибки. Много ошибок. В обычной жизни они не столь уж опасны, поскольку обычная наша деятельность толерантна к незначительным проступкам. Можно нажимать вилкой на блюдо сильнее, чем требуется, пить быстрее, чем полагается, выжимать педаль сцепления резче, чем нужно, использовать более крепкие выражения, чем полагается. Это случается, и в большинстве случаев это не мешает вам достичь желаемой цели: есть, пить, вести машину, общаться с друзьями.

Но в программировании все обстоит по-другому. С невероятной скоростью компьютер будет выполнять программу, задающую вашу машину так, как вы ее создали. Компьютер «не понимает» вашей программы, он просто выполняет ее, малейшая ошибка тут же отразится на механизмах машины. Что написали, то и получите.

Это правило, пожалуй, главное, что всегда следует помнить, имея дело с компьютером и обучаясь программированию. Возможно, до сих пор вы верили в другое, в то, что компьютеры умны, ведь программы кажутся столь совершенными. Примером подобного чуда является поиск в Интернете, благодаря которому менее чем за секунду можно найти среди множества предложений идеальное место для вашего отдыха. Хотя человеческий разум встроен во многие программы, компьютер, их выполняющий, подобен преданному и не рассуждающему слуге, бесконечно честному, бесконечно быстрому и определенно глупому. Он будет беспрекословно выполнять данные ему инструкции, никогда не проявляя инициативы исправить ошибки, хотя человек нашел бы их очевидными. Ответственность на вас – программисте, вы должны предоставить этому покорному созданию безупречные инструкции, задающие — при выполнении любой важной программы — миллиарды элементарных операций.

Работая с компьютером, вы, вероятно, замечали, что он не всегда функционирует так, как хотелось бы. Бывает, что компьютер «зависает», или «падает», — все перестает работать. За исключением редких случаев отказа аппаратуры, это не компьютер «зависает», а программа «подвесила» его, то есть его «подвесил» программист, не предусмотревший все возможные сценарии работы.

Вы не сможете научиться программированию, не получив подобного опыта работы – вашего собственного опыта, не убедившись на практике, что не все работает, как должно; вы не станете профессиональным программистом, если не овладеете методиками, позволяющими вам строить программы, которые выполняют работу так, как вы хотите.

Есть хорошая новость: такие программы можно создавать, овладев подходящим инструментарием и дисциплиной сопровождения, обращая внимание как на всю картину в целом, так и на детали.

Помочь вам овладеть предметом – одна из главных задач этой книги, которая является не просто введением в программирование, но учит программировать хорошо. Заметьте: начиная со следующей главы появятся советы «Почувствуй методологию» и «Почувствуй стиль», где собраны рекомендации– плоды долгих лет работы, результаты трудного пути, которые помогут вам создавать ПО, работающее так, как вы того желаете.

1.2. Общие установки

В следующей главе мы перейдем непосредственно к разработке программ. Нам не понадобится детальное знание компьютера, но давайте ознакомимся с его фундаментальными свойствами, поскольку они устанавливают контекст для конструирования программ.

В главе 10 «Немного об аппаратуре» даются дополнительные сведения о работе компьютера.

Задачи компьютеров

Компьютеры – *цифровые компьютеры с автоматически хранимой программой*; если быть более точным — это машины, которые могут хранить и находить информацию, выполнять операции над этой информацией и обмениваться информацией с другими устройствами.

Следующее определение подчеркивает основные способности компьютера.

Компьютеры выполняют

- хранение и поиск
- операции
- коммуникации

Хранение и доступ – необходимое условие для всего остального: компьютеры должны иметь возможность где-то хранить и откуда-то извлекать информацию, прежде чем они смогут ее обрабатывать или передавать. Это «где-то» называется **памятью**.

Операции включают сравнение («Являются ли два значения одинаковыми?»), замену («Замени одно значение на другое»), арифметику («Вычисли сумму этих двух значений») и другое. Операции являются примитивными, но, тем не менее, компьютеры способны совершать изумительные вещи благодаря скорости, с которой они выполняют базисные операции и мастерству человека – вашему, того, кто пишет программы.

Коммуникация позволяет нам вводить информацию в компьютер и находить нужную нам информацию (оригинальную либо созданную в процессе выполнения операций). Компьютеры могут также общаться с другими компьютерами и различными устройствами, такими как датчики, телефоны, дисплеи и многие другие.

Общая организация

Предыдущее определение дает основу для построения следующей диаграммы.

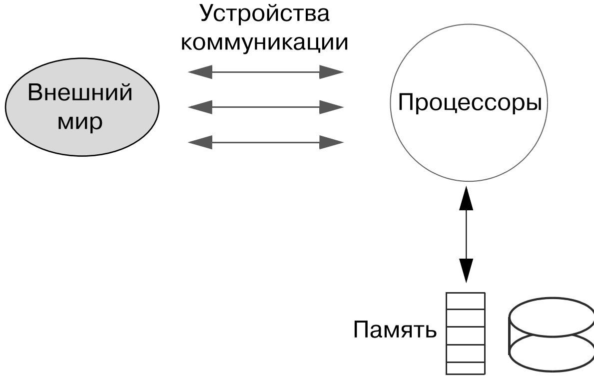


Рис. 1.2. Компоненты компьютерной системы

Память хранит информацию. Говоря о памяти, нужно понимать, что у компьютера могут быть несколько устройств, называемых памятью, и они могут быть разного вида, отличаясь размерами, скоростью доступа и живучестью (возможностью сохранения информации при отключении энергии).

Процессоры выполняют операции. И снова их может быть несколько. Обычно мы встречаемся с процессором, называемым ЦПУ (CPU, аббревиатура для устаревшего термина *Central Processing Unit*).

Устройства коммуникации обеспечивают способ взаимодействия с внешним миром. На рисунке показана связь внешнего мира с процессором, а не непосредственно с памятью; в действительности, когда нужно изменить информацию в памяти при вводе данных из внеш-

него мира, предварительно нужно выполнить некоторые операции процессора. Устройства коммуникации поддерживают либо **ввод** (из мира в компьютер), либо **вывод** (в обратном направлении), либо обмен в обоих направлениях. Примерами устройств являются:

- клавиатура, с помощью которой человек вводит тексты (ввод);
- дисплей или терминал (вывод);
- мышь или джойстик, позволяющий создавать точки на экране (ввод);
- датчики, регулярно посылающие результаты измерений производственных параметров в компьютер (ввод);
- сетевое соединение, позволяющее связываться с другими компьютерами и устройствами (ввод-вывод).

Аббревиатура **I/O** используется для ввода и вывода**.**

Информация и данные

Ключевое слово в определении компьютеров – *информация*, которую мы храним в памяти, обрабатываем и которой обмениваемся, используя устройства коммуникации.

Это верно с точки зрения человека. Строго говоря, компьютеры не манипулируют с информацией, они имеют дело с *данными*, представляющими информацию.

Определения: данные, информация

> Совокупности символов, хранящиеся в компьютере, называются данными. Любая интерпретация данных в интересах человека называется информацией.

Информация может быть всем, чем угодно: заголовками новостей, фотографией друга, тезисами докладчика на семинаре. Данные — закодированная форма информации.

Приведу пример: аудиоформат MP3 задает множество правил, позволяющих декодировать информацию о музыкальном произведении, которое может храниться в виде данных в памяти компьютера, может быть передано по сети и послано аудиоустройству для воспроизведения.

Данные хранятся в памяти. Задача устройств коммуникации — создать данные из приходящей информации, сохранить их в памяти и, когда процессоры преобразуют эти данные и создадут новые, передать их во внешний мир в таком виде, чтобы данные воспринимались как информация. В адаптированном виде этот процесс показан на следующей картинке:

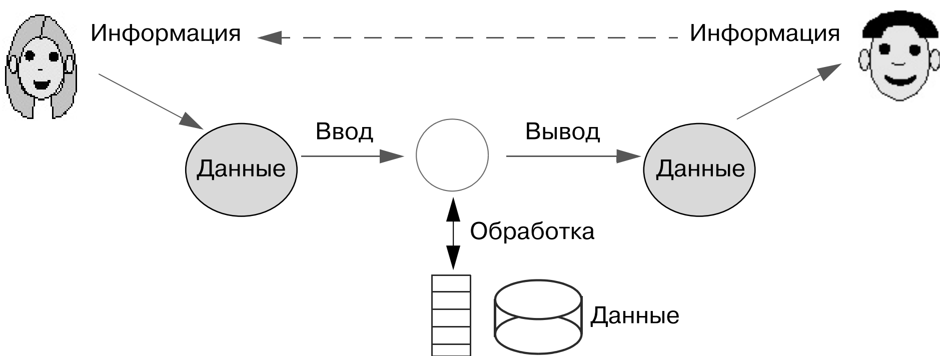


Рис. 1.3. Обработка информации и данных

Стрелки, идущие вправо и влево, показывают, что процесс не однонаправленный, а повторяющийся, и благодаря обратной связи позволяет вырабатывать новые результаты.

Компьютеры повсюду

Обыденной картиной является настольный или портативный компьютер, чей процессор и память могут быть упрятаны в ящик размером с учебник, подобный этой книге, или с большой словарь. Они имеют размеры, соответствующие удобству человека. Среди «ручных» размеров встречаются такие устройства, как мобильный телефон, являющийся сегодня карманным компьютером с расширенными функциями коммуникации. Для сложных научных вычислений (физика, предсказание погоды...) применяются суперкомпьютеры, размеры которых соизмеримы с размерами комнаты. Конечно, эти размеры не идут ни в какое сравнение с размерами компьютеров предыдущих поколений, занимавших целые здания при более чем скромных возможностях.

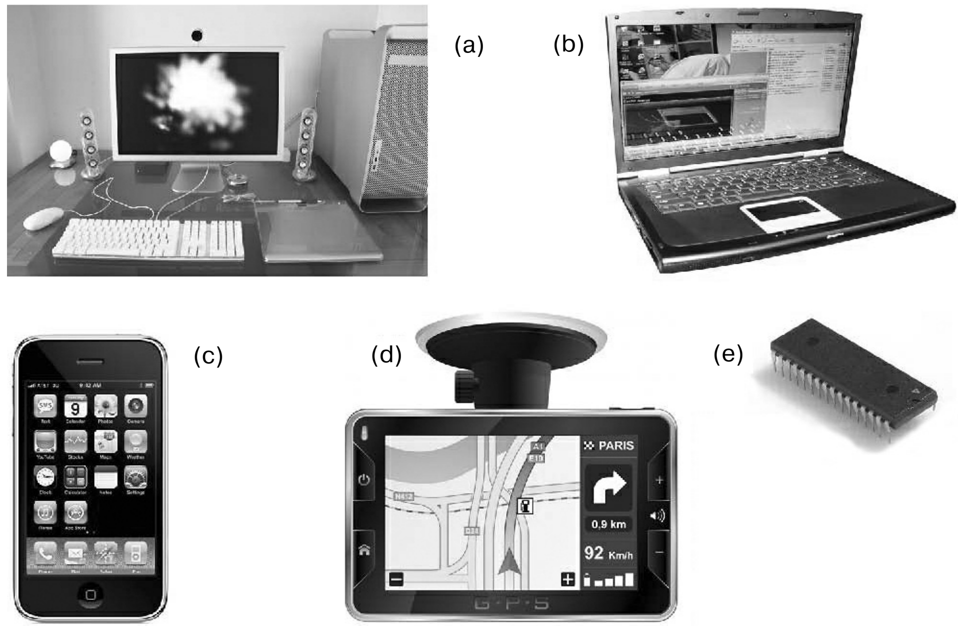


Рис. 1.4. Компьютеры: (a)настольный; (b) портативный; (c) iPhone (Apple); (d) навигационная система GPS;(e) встроенный процессор

Размеры процессора и памяти значительно меньше, чем занимают показанные на рисунке устройства. Растет число компьютеров, **встраиваемых** в бытовые устройства. Современные автомобили напичканы компьютерами, регулирующими подачу топлива, торможение, даже открытие окон. Принтер, связанный с компьютером, сам является компьютером, способным создавать шрифты, сглаживать изображения, после заминки бумаги начинать печать с последней необработанной страницы. Электрические бритвы включают компьютеры. Стиральные машины содержат компьютеры, и в ближайшем будущем маленькие компьютеры могут встраиваться в одежду для согласования с процессом стирки.

Компьютеры, которые вы будете использовать для выполнения упражнений этой книги, — это обычные компьютеры с клавиатурой, мышкой, дисплеем, но подсознательно нужно всегда иметь в виду, что методика разработки ПО покрывает более широкую область. ПО для встроенных систем должно удовлетворять более жестким требованиям: неверное срабатывание системы торможения может привести к ужасным последствиям, и их нельзя исправить, как при выполнении программ на настольном компьютере, остановив выполнение, исправив ошибку и запустив программу повторно.

Компьютер с хранимой программой

Компьютер, как отмечалось, является универсальной машиной, способной выполнять любую введенную в него программу.

> Для процесса ввода используются устройства коммуникации, обычно клавиатура и мышь. Текст появляется на экране при его печати и кажется, что это непосредственный результат ввода, но это иллюзия. Клавиатура является устройством ввода, дисплей – устройством вывода, отображение входного текста на экране требует специальной программы, называемой текстовым редактором, получающей ввод, обрабатывающей его и выводящей результат на экран. Благодаря скорости работы компьютера возникает иллюзия непосредственной связи клавиатуры и дисплея.

Когда программа вводится, куда же она попадает? В память, доступную для ее хранения. Вот почему говорят о компьютере с хранимой программой. Чтобы стать специализированной машиной, способной выполнять специфические задачи, которые вы как программист поставили ему, компьютер будет читать ваши приказы из своей памяти.

Свойство хранимой программы объясняет, почему мы не дали подходящего определения для устройств памяти. Следовало бы сказать, что память — это устройство для хранения и доступа к данным, но тогда понятие данных распространялось бы и на программы. Разумнее разделять эти два понятия.

Определение: память

Память – устройство для хранения и доступа к данным и программам.

Способность компьютеров рассматривать программы как данные — *выполняемые данные* — объясняет их исключительную гибкость. В начале компьютерной эры это привело к осознанию существования самомодифицируемых программ (так как программы могут изменять данные, они могут и модифицировать программы, включая и саму исполняемую программу). Отсюда пошли философские рассуждения, что в результате последовательности самомодификаций программы могут стать умнее создателя и компьютер захватит власть над миром.

Действительность, с которой мы сталкиваемся сегодня, — более прозаическая и более неприятная. Например, одна из причин, почему пользователи электронной почты должны быть крайне осторожны при открытии вложений, приходящих с письмом, состоит в том, что приходящие данные могут оказаться зловредно написанной программой, чье выполнение разрушит другие данные.

Для программистов свойство хранимой программы имеет непосредственное следствие: оно делает программы подлежащими, подобно данным любого другого типа, различным трансформациям, выполняемым другими программами. В частности, *программа, которую вы пишете, это не та программа, которую вы выполняете*. Операции, которые может выпол-

нять процессор, спроектированы для машин, не для человека, их непосредственное использование утомительно и чревато ошибками. Вместо этого вы будете:

- писать программы в нотации, спроектированной для людей и называемой *языками программирования*. Эта форма программы называется исходным кодом (исходным текстом, иногда просто исходником);
- зависеть от специальных программ, называемых компиляторами, которые преобразуют программу, читаемую человеком, в формат, подходящий для выполнения процессором (этот формат называют машинным кодом, объектным кодом).

Мы часто будем сталкиваться со следующими терминами, отражающими это разделение задач.

Определения: статика, динамика

Статические свойства программы – это свойства исходного текста, которые могут быть проанализированы компилятором.
Динамические свойства – это те свойства, которые характерны для этапа выполнения машинного кода на компьютере.

Детали всего этого — коды процессора, языки программирования, компиляторы, примеры статических и динамических свойств — появятся в последующих главах. На данный момент следует знать, что программы, которые вы собираетесь писать, начиная со следующей главы, предназначены как для людей, так и для компьютеров.

Человеческий аспект программирования является центральным для инженерии программ. Когда вы программируете, вы пишете текст не только для компьютера, но и для человека, например, для того, кто позже будет читать вашу программу, чтобы добавить в нее новые функции, исправить ошибки. Это хороший повод, чтобы позаботиться о читабельности вашей программы. Дело не только в том, чтобы быть любезным по отношению к другому человеку; этим «другим» можете быть вы сами, когда, став старше на несколько месяцев, будете пытаться расшифровать написанное в исходной версии и чертыхаться, силясь понять, чтобы это могло значить.

В этой книге внимание акцентируется не только на приемах, которые делают программу хорошей для компьютера (таких как *эффективность* выполнения, позволяющих достаточно быстро выполнять программу), но и на приемах, которые делают программу хорошей для чтения человеком. Программные тексты должны быть *понятными*, *расширяемыми* (простыми для внесения изменений); программные элементы должны быть повторно используемыми, чтобы при повторной встрече с похожей задачей не пришлось бы повторно изобретать ее решение. Программы должны быть *устойчивыми*, защищать себя от ошибочно введенных данных. И главное, они должны быть *корректными*, выдавать ожидаемые результаты.

Программистский фольклор: все важное в дырочках

Ветераны аэрокосмической индустрии рассказывают историю об одном инженере в эпоху первых космических полетов, который был ответственным за вес всего, что попадает на борт космического корабля. Он приставал к программистам, требуя, чтобы они назвали вес управляющего ПО. Ответ неизменно был, что ПО вообще ничего не весит; но инженера это не убеждало.

Однажды он пришел к главному программисту, размахивая пачкой перфокарт (средство ввода данных в те времена, смотри рисунок). «Это и есть ПО, – закричал он, – разве я не говорил вам, что и оно, подобно всему, имеет вес!» На что программист, не раздумывая, ответил ему: «Видите дырки? Они и есть ПО».

(Возможно, апокриф, но все же хорошая история)

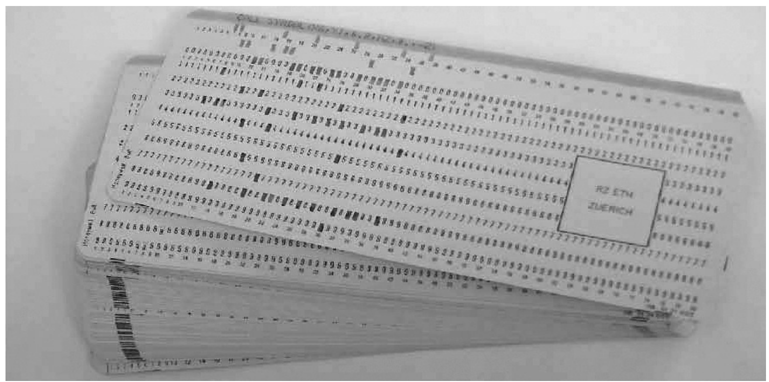


Рис. 1.5. Колода пробитых перфокарт

1.3. Ключевые концепции, изученные в этой главе

- *Компьютеры* – машины общецелевого назначения. Компьютер с загруженной программой превращается в машину специального назначения.
- Компьютерные программы обрабатывают, хранят и передают *данные*, представляющие *информацию* в интересах человека.
- Компьютер состоит из *процессоров*, *памяти* и *коммуникационных устройств*. Все вместе называется аппаратурой (*hardware*).
- Программы и связанные с ними интеллектуальные ценности называются ПО (*software*). ПО является инженерным продуктом чисто интеллектуальной природы.
- Программы должны быть сохранены в памяти до начала их выполнения. Они могут существовать в разных формах, некоторые из которых предназначены для человека и хранятся в читаемой форме, другие – непосредственно обработаны, чтобы их мог выполнять процессор.
- Компьютеры появляются в различных обликах: многие *встраиваются* в товары и устройства.
- Программы должны быть написаны так, чтобы их легко было понимать, расширять и повторно использовать. Они должны быть корректными и устойчивыми.

Новый словарь

В конце каждой главы вы найдете такой список. Проверьте (и это есть первое упражнение главы), знаете ли вы смысл каждого введенного термина, найдите его определение.

Communication device	Устройство коммуникации	**Compiler**	Компилятор
Correct	Корректный	**Computer**	Компьютер
Data	Данные	**CPU**	ЦПУ
Embedded	Встроенный	**Dynamic**	Динамический
Hardware	Аппаратура	**Extendible**	Расширяемый
Input	Ввод	**Information**	Информация
Output	Вывод	**Memory**	Память
		Persistence	Живучесть

Processor	Процессор	**Programmer**	Программист
Programming language	Язык программирования	**Reusable**	Повторно используемый
Robust	Устойчивый	**Software**	ПО – программное обеспечение
Source	Исходный код	**Static**	Статический
Target	Целевой, машинный код	**Terminal**	Дисплей, монитор
User	Пользователь		

1-У. Упражнения

1-У.1. Словарь

Дайте точное определение каждого термина из словаря.

1-У.2. Данные и информация

Для каждого из следующих предложений скажите, характеризует ли оно данные, информацию или то и другое (поясните решение):

1. «Вы можете найти детали рейса на сайте».
2. «Когда печатаете в этом поле, используйте не более 60 символов на строку».
3. «Ваш пароль должен содержать не менее 6 символов».
4. «У нас нет данных о вашей оплате».
5. «Вы не сможете оценить ее сайт, не подключив Flash».
6. «Это прекрасно, что вы дали мне ссылку на ваш сайт, но я, к сожалению, не могу читать по-итальянски!»
7. «Это прекрасно, что вы дали мне ссылку на ваш сайт, и мне хотелось бы читать по-русски, но мой браузер отображает кириллицу как мусор».

1-У.3. Дайте точное определение того, что вы хорошо знаете

Все хорошо знают алфавитный порядок, который еще называют лексикографическим порядком. В соответствии с этим порядком расположены слова в словарях и других упорядоченных по алфавиту списках. Для любых двух слов языка этот порядок устанавливает предшествование, какое слово встретится в словаре раньше другого. Так, слово «кожа» предшествует слову «кора», которое, в свою очередь, предшествует слову «корабль». В упражнении требуется:

Определить условия, при которых одно слово предшествует другому.

Другими словами, требуется дать определение лексикографического порядка. Это понятие вы, несомненно, умеете применять на практике. Упражнение требует точного определения интуитивно ясного понятия. Оно может понадобиться, если вам требуется, например, написать программу, составляющую упорядоченные списки.

Чтобы сконструировать определение, вы можете предполагать, что:

- **слово** является последовательностью из одной или нескольких букв (годится и предположение, что слово может быть пустым, не содержащим ни одной буквы). Укажите, какое определение слова вы выбрали;

- **буква –** это элемент конечного множества (алфавита);
- выбор алфавита не имеет особого значения. Важно только, что все буквы из алфавита уже упорядочены, так что для каждой пары букв алфавита известно, какая из них предшествует (**меньше**) другой.

Можно, например, воспользоваться прописными буквами латиницы — буквами, записанными в нижнем регистре, порядок для которых известен: *a b c d e f g h i j k l m n o p q r s t u v w x y z*.

Проблема в том, что требуется определение, а не рецепт. Не годится решение в форме: «Сравним первые буквы двух слов. Если первая буква первого слова меньше первой буквы второго слова, то первое слово предшествует второму, в противном случае будем ...» Это рецепт, а не определение. Определение может иметь следующую форму: «*Слово* w1 *предшествует слову* w2*, если и только если выполняется одно из следующих условий* ...»

Убедитесь, что ваше определение покрывает все возможные случаи и соответствует интуитивным свойствам лексикографического порядка, например, невозможно, чтобы для различных слов *w*1и *w*2 слово *w*1предшествовало *w*2 и слово *w*2 предшествовало *w*1.

Об этом упражнении. Цель – применить вид точных, не операционных знаний при конструировании ПО. Идея заимствована у известного датского ученого, основоположника структурного программирования Эдсгера Дейкстры.

1-У.4. Антропоморфизм

Сопоставьте компоненты и функции компьютерной системы с частями и функциями человеческого тела. Обсудите схожесть и различие.

2

Работа с объектами

Теперь мы переходим к написанию, выполнению и модифицированию нашей первой программы.

Предварительные требования: вы должны владеть основами работы с компьютером, уметь работать с каталогами (папками) и файлами. Среда разработки EiffelStudio должна быть инсталлирована на вашем компьютере и должна быть загружена система Traffic. Все остальные знания можно получить из этой книги и на поддерживающем сайте.

Загрузка Traffic: traffic.origo.ethz.ch. Сайт книги: touch.ethz.ch.

2.1. Текст класса

Некоторые главы этой книги поддерживаются «программными системами» (проектами — коллекцией файлов, составляющих систему), включенными в поставку Traffic. В имени системы отражается название главы (английское): *объекты*, *интерфейсы*, *создание*. В каталоге *example* данной поставки каждая система появляется в подкаталоге, имя которого также включает номер главы: *02_object, 04_interfaces* и так далее.

Запустите EiffelStudio и откройте «систему», названную *objects*. Точные детали того, как это делается, даны в приложении, описывающем среду разработки EiffelStudio.

Смотри «Установки проекта», E.2.

Почувствуй практику

Использование EiffelStudio

Так как эта книга фокусируется на принципах конструирования ПО, детали того, как использовать инструментальные средства EiffelStudio для выполнения примеров, вынесены в приложение E: «Использование среды разработки EiffelStudio». Они доступны и на соответствующей веб-странице. Прочтите это приложение.

В случае если что-то пойдет не так, помните следующий совет.

Почувствуй практику

Если вы запутались

Если у вас нет опыта работы с компьютером, вполне вероятно, что вы совершите ошибку, уводящую вас от столбовой дороги. Старайтесь избегать этой ситуации, следуя в точности предписаниям, но если это все же случится, не паникуйте и еще раз проанализируйте приложение E.

Программные тексты, появляющиеся в книге, и те, что вы будете видеть на экране компьютера при работе в EiffelStudio, немного отличаются из-за различия возможностей печатной книги и компьютера.

Начнем работу с элементом программы – классом, названным *PREVIEW* (предварительный просмотр), составляющим ядро нашей первой программы. Скопируйте текст этого класса.

Смотри снова: Е.2, как копировать класс.

Отображение на дисплее должно выглядеть так:

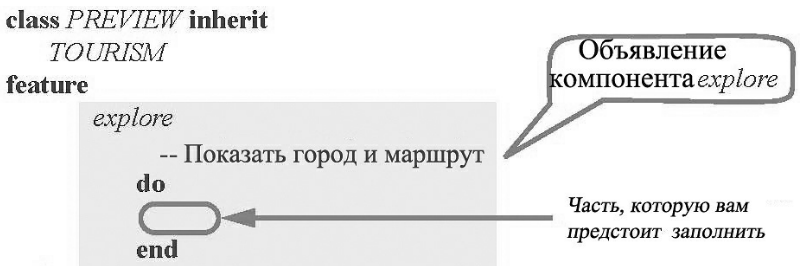


Рис. 2.1.

Первая строчка говорит, что мы смотрим на «класс», одну из фундаментальных машин, из которых строится программа. Он назван *PREVIEW*, поскольку класс описывает небольшую часть просмотра путешествия по городу.

Первые две строки устанавливают также, что *PREVIEW* – **наследник** существующего класса *TOURISM*. Это означает, что *PREVIEW* расширяет *TOURISM*, имеющий много полезных свойств. Все, что вам нужно сделать, так это добавить свои собственные идеи в новый класс *PREVIEW*. Имена классов отражают отношение между ними: *TOURISM* описывает общее понятие туров по городу; *PREVIEW* рассматривает частный вид тура. Речь не идет о настоящем визите, а о предварительном просмотре, который можно выполнить, сидя с комфортом за своим столом.

Почувствовали магию?

Класс TOURISM является частью ПО, подготовленного специально для этой книги. Не строя все с нуля, а пользуясь комбинациями предопределенных свойств, вы сможете непосредственно изучить общие программистские концепции и получить практику, выполняя примеры программ.

Может показаться, что первая программа подобна магии. Это не так. Поддерживающее ПО – кажущееся магией – использует методы, изучаемые в этой книге. Шаг за шагом мы будем снимать пелену магии, в конце все станет явным, и вы сами сможете провести реконструкцию по вашему желанию.

Даже теперь ничто не мешает вам просмотреть поддерживающее ПО, например, класс TOURISM, поскольку все открыто. Но не ожидайте, что все сразу будет понятно.

Текст класса в общем случае описывает множество операций. Описание каждой операции является частью класса — его *компонентом* (*feature*). В данном примере есть только один компонент, названный *explore*. Часть класса с его описанием называется **объявлением (declaration)** компонента. Объявление включает:

- имя компонента – *explore*;
- *комментарий*: «—Показать город и маршрут»;
- фактическое содержимое компонента, его *тело*, заключенное между *ключевыми словами* **do** и **end** — оно в данный момент пустое, но его предстоит заполнить.

Ключевое слово – это зарезервированное слово со специальным смыслом; его нельзя использовать для именования собственных классов и компонентов. Чтобы отличать ключевые слова, их всегда выделяют жирным шрифтом (**bold**). В данном тексте ключевыми словами являются: **class, inherit, feature, do** и **end.** Этих пяти слов хватит надолго.

Комментарий, такой как «—Показать город и маршрут», является поясняющим текстом, не влияющим на выполнение программы, но помогающий *людям* понять программный текст. Всюду в тексте класса два подряд идущих дефиса – два символа «—», задают начало комментария, продолжающегося до конца строки. **Всегда** по правилам хорошего стиля в объявление компонента класса следует сразу же после первой строчки включать комментарий, объясняющий роль компонента, как это здесь и сделано.

2.2. Объекты и вызовы

Ваша первая программа позволит подготовить путешествие по городу, замечательному Парижу. Наше путешествие будет безопасным: все, что мы хотим сделать, — отобразить некоторую информацию на экране.

- Прежде всего, отобразить карту Парижа, включая карту метро.
- Во-вторых, подсветить на карте расположение музея Лувр, о котором вы все, конечно, слышали.
- Затем подсветить на карте одну из линий метро – линию 8.
- Наконец, поскольку ваш предусмотрительный агент подготовил маршрут для первого путешествия, выполнить анимацию этого маршрута, показывая небольшое изображение путешественника, прыгающего на каждой остановке.

Редактирование текста

Настало время программирования!

Создадим первую программу

В этом разделе вас попросят заполнить текст вашей первой программы, а затем выполнить ее.

Необходимо отредактировать текст класса *PREVIEW*, модифицировав компонент *explore*. Его текст должен выглядеть так:

Во избежание недоразумений при вводе текста следуйте следующим правилам.

- Текст каждой строчки начинается на некотором расстоянии от левой границы. Отступы служат для отображения структуры текста. Поскольку они не влияют на выполнение, можно было бы выравнивать текст по левой границе, но это существенно ухудшило бы понимание текста (и, вероятно, вашу оценку за предоставленную программу), так что, пожалуйста, тщательно следуйте этому правилу. Правила отступа будут даны далее.

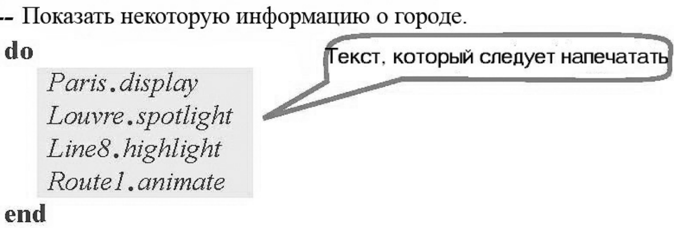


Рис. 2.2.

- Для достижения отступа не используйте повторяющиеся пробелы, который могут внести путаницу в выравниваемый текст; применяйте символ табуляции *Tab*.
- В *Paris.display* и далее на следующих строчках вы видите символ «.», разделяющий два слова. В отличие от точки, завершающей предложение, за ней не следует пробел.

Заметьте, что фактически не нужно все печатать. В EiffelStudio встроен механизм дополнения, который предлагает список возможных продолжений начального текста, напечатанного вами. Как только вы поставите точку, напечатав «*Paris.*», студия предложит выпадающее меню — список возможностей – список тех компонентов, что применимы к *Paris*. Можно прокрутить этот список и добраться до компонента *display*. Если список длинный, то проще напечатать одну или две первые буквы имени компонента, например *d* от *display*; тогда список обновится, и вы сразу же попадете на имя, начинающееся с *d*:

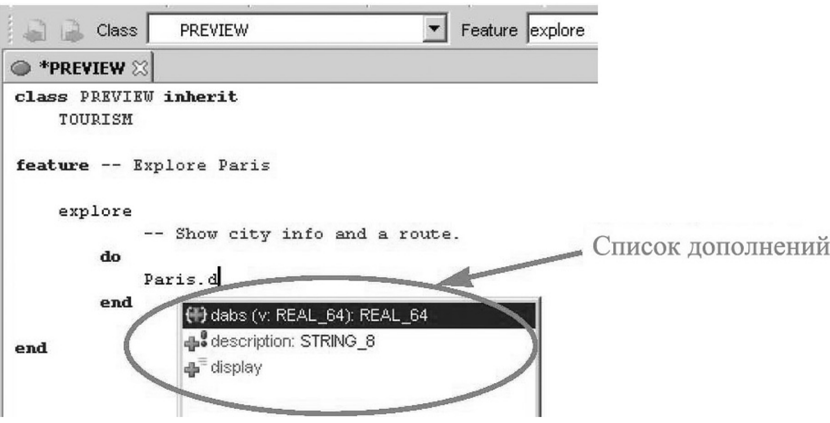


Рис. 2.3.

Меню дополнений автоматически появляется в некотором контексте, как мы уже видели в случае с напечатанной точкой. Если же меню не появляется, а помощь вам требуется, то нажмите пару «горячих» клавиш CTRL + Space.

После внесения изменений стоит их сохранить, для чего выберите в меню File пункт Save или нажмите Control +S. Если вы забудете сохранить изменения, студия напомнит об этом в нужный момент.

Выполнение первой программы

Запустим программу на выполнение. Подробности можно найти в приложении. Все, что нужно, — это нажать кнопку Run, расположенную справа в верхней части окна. В первый раз, когда вы сделаете это, появится диалоговое окно, предупреждающее, что программу перед выполнением необходимо предварительно скомпилировать:

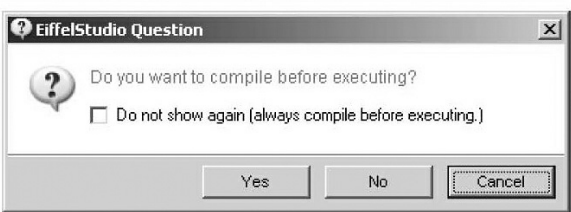


Рис. 2.4.

«Компиляция» системы, написанной вами и представленной в форме исходного кода, означает преобразование ее в форму, которая непосредственно может быть обработана компьютером.

Это полностью автоматический процесс. Некоторые предпочитают забывать о компиляции, неявно полагая, что компьютер выполняет непосредственно написанную ими программу. Многие ставят флажок в появившемся диалоговом окне, чтобы избежать появления напоминания о компиляции в будущем. Лично я не включаю этот флажок, поскольку предпочитаю явно запускать компиляцию программы, нажимая на кнопку Compile или горячую клавишу F7. Свои предпочтения оформите позже, а в первый раз ответьте «Yes» в диалоговом окне.

Если вы забыли сохранить изменения, студия обнаружит это и предложит сообщение, подобное предыдущему. Здесь снова можно включить флажок, позволяющий автоматически сохранять изменения перед запуском.

Компиляция стартует. Студия должна скомпилировать не только класс *PREVIEW* с его единственным компонентом *explore*, но и все, что необходимо, — возможно, всю систему Traffic и поддерживающие библиотеки. Это зависит от того, установлена ли на вашем компьютере уже скомпилированная версия. Полная компиляция занимает время, но это делается только один раз. Все последующие компиляции будут компилировать только сделанные изменения, так что будут выполняться практически мгновенно, даже если вся программа большая.

Если вы что-то не так напечатали, появятся сообщения об ошибках, которые можно исправить и снова запустить процесс компиляции и выполнения.

Вы увидите следующую последовательность событий.

1. В результате выполнения первой строки компонента *explore* - *Paris.display* -появится карта города, включающая структуру линий метро:
2. В течение 5 секунд все останется без изменений, а затем, как результат действия второй строки, *Louvre.spotlight*, подсветится музей Лувр, расположенный рядом со станцией метро Palais Royal:

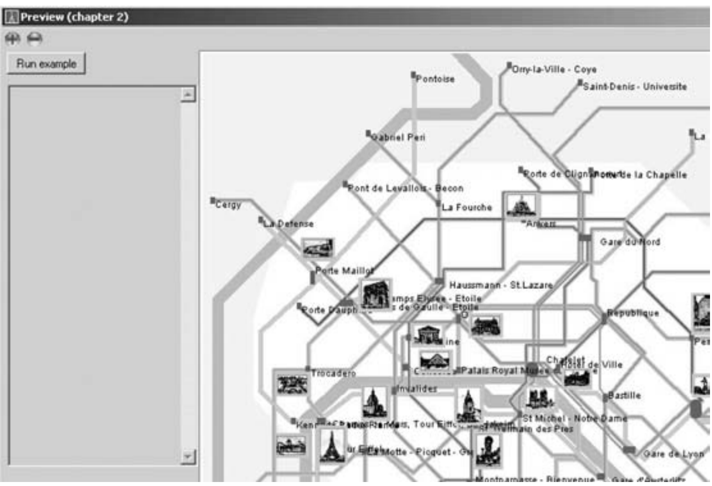


Рис. 2.5.

Рис. 2.6.

3. Еще через 5 секунд подсветится линия 8 в сети метро, как результат выполнения третьей строки нашего компонента, *Line8.highlight*:

Рис. 2.7.

4. После очередной короткой задержки выполнение 4-й строки, *Route1.animate,* приведет к появлению человечка, передвигающегося вдоль линии метро по выбранному маршруту.

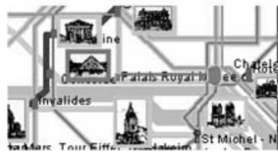


Рис. 2.8.

Однажды подготовленная программа может выполняться многократно по вашему желанию. Если вы не измените программу, она будет просто запускаться на выполнение. Если же вы измените ее, EiffelStudio перекомпилирует ее, запросив подтверждения, при условии, что вы не включили автоматическую компиляцию при изменениях. После компиляции измененную программу можно снова запустить на выполнение.

Анализ программы

Выполнение программы является результатом действия 4-х строк, вставленных в текст компонента *explore*. Давайте посмотрим, каков их точный смысл. Техника, использованная в этой простой программе, фундаментальна, поэтому убедитесь, что вы все понимаете в следующих ниже объяснениях.

Первая строка

```
Paris.display
```

использует **объект**, известный программе как *Paris*, и **компонент**, известный как *display*.

Объект является единицей данных (в следующем разделе это понятие обсуждается в деталях), компонент есть операция, применимая к этим данным. *Paris.display* — фундаментальная программистская конструкция, известная как вызов компонента:

```
x.f
```

где *x* обозначает объект, а *f* — компонент (операцию). Эффект вызова хорошо определен:

Почувствуй семантику

Вызов компонента

Эффект времени выполнения вызова компонента *x.f* состоит в применении компонента с именем *f*, принадлежащего соответствующему классу, к объекту, который обозначает *x* в момент выполнения.

Предыдущие правила касались формы или синтаксиса программы. Здесь мы имеем дело с первым правилом, определяющим семантику – поведение программы в период выполнения. Мы будем изучать эти правила в деталях в последующих главах.

Вызов компонента — основа вычислений, повторяю еще и еще раз — это то, что делают наши программы во время выполнения.

В нашем примере целевой объект называется *Paris*. Как следует из его имени, он представляет город. Как много от настоящего города «Парижа» содержится в его описании? Не

стоит об этом беспокоиться, так как *Paris* для нас является предопределенным объектом. Довольно скоро мы научимся определять свои собственные объекты, но в данный момент вы должны основываться на тех объектах, что подготовлены для вас в этом упражнении. Стандартные соглашения позволяют распознавать такие объекты:

Почувствуй стиль

Имена предопределенных объектов

Имена предопределенных объектов всегда начинаются с буквы в верхнем регистре, как например *Paris, Louvre, Route1*.

Имена определяемых вами объектов должны начинаться с буквы в нижнем регистре (по правилам стиля).

Где же определены эти предопределенные объекты? Вы уже догадались – в классе *TOURISM*, который **наследует** ваш класс *PREVIEW*. Это и есть то место, куда мы поместили «магию», благодаря которой ваша программа, простая сама по себе, может получать важные и интересные результаты.

Одним из компонентов класса, применимых к объекту, который представляет город, такой как *Paris*, является *display*. Этот компонент ответственен за показ на экране текущего состояния города.

После применения *display* к объекту *Paris* программа выполнит следующий вызов компонента:

```
Louvre.spotlight
```

Целевым объектом здесь является *Louvre*, еще один предопределенный объект (его имя начинается с заглавной буквы), обозначающий музей Лувр. Компонентом служит *spotlight*, подсвечивающий соответствующее место на карте.

Затем, для подсветки 8-й линии метро, мы выполняем следующий вызов:

```
Line8.highlight
```

Компонент *highlight* в соответствии со своим названием подсвечивает целевой объект – объект, вызвавший метод, в данном случае *Line8*, представляющий линию метро с номером 8.

На заключительном шаге еще один предопределенный объект вызывает свой компонент:

```
Route1.animate
```

Целевым объектом является *Route1*, представляющий предопределенный маршрут, – можно считать, что он был подготовлен вашим турагентом. Компонент *animate* будет показывать маршрут, передвигая фигурку по ходу движения.

Для того чтобы программа работала, как ожидается, используемые в ней компоненты – *display*, *spotlight*, *highlight*, *animate* – должны делать немного больше, чем просто отображать что-то на экране. Причина в том, что компьютеры быстры, *очень* быстры. Так что, если бы единственным эффектом выполнения операции *Paris*.*display* было отображение карты Парижа, то следующая операция *Louvre*.*spotlight* выполнялась бы через мгновение после первой и вы бы не успели заметить на экране результат первой операции, показывающий карту Парижа без подсветки Лувра. Во избежание этого все компоненты устроены так, что после показа на экране того, что они должны показать, в исполнении программы наступает пауза, длящаяся 5 секунд.

Все это делается в тексте вызываемых компонентов, который мы пока не рассматриваем (хотя при желании вы можете его увидеть).

Примите поздравления! Вы написали и выполнили первую программу и, более того, поняли, что она делает.

2.3. Что такое объект?

Наш пример программы работает с объектами – четыре из них называются *Paris*, *Louvre*, *Line8* и *Route1*. Работа с объектами – это главное, что делают все программы. Понятие объекта настолько фундаментально, что дало имя самому стилю программирования, используемому в этой книге и широко применяемому в программной индустрии: Объектно-Ориентированное, часто заменяемое короткой аббревиатурой, ОО-программирование.

Объекты, которые можно ударить, и те, которые бить невозможно

Что следует понимать под словом «объект»? Мы используем для технических целей слово из обычного языка – вполне обыденного языка, ибо трудно придумать более общее понятие, чем объект. Каждый может дать непосредственное толкование этого понятия, в этом есть как преимущество, так и предмет потенциального непонимания.

Польза в том, что использование объектов в нашей программе позволяет организовать ее как модель реальной системы с реальными объектами. Когда вам доведется быть в Париже, вы убедитесь, что Лувр – реальный объект; если взгляд вас не убедит, то можно подойти и «пнуть» его пяткой (учтите, покупка этой книги не дает права на оплату медицинских расходов). Наш второй объект, *Paris*, столь же реален в жизни и представляет целый город.

Но удобство использования программных объектов для представления физических объектов не должно приводить к недоразумениям двух видов. Реальность программного объекта не распространяется далее чем на нематериальную коллекцию данных, хранящихся в памяти компьютера. Наша программа может установить их так, чтобы операции над данными моделировали операции над физическим объектом. Например, *Bus48*.*start* представляет операцию, приводящую автобус в движение, — но эта связь мысленная, существующая в воображении. Хотя наша программа использует объект *Paris*, но это не настоящий Париж. «Никто не может поместить Париж в бутылку», — говорит старая французская поговорка, и вам не удастся поместить Париж в программу.

Никогда не забывайте, что слово «объект», используемое в этой книге, означает программное понятие. Некоторые программные объекты имеют аналоги во внешнем мире, но это не обязательно, как мы увидим, переходя к более сложным программистским приемам. Даже наш последний в четверке объект *Route1* представляет маршрут – мысленный план путешествия. Этот план предполагает поездку на метро от станции Лувр (также известной как «Пале-Рояль») до станции Сен-Мишель. Как показано на рисунке, этот маршрут имеет три этапа.

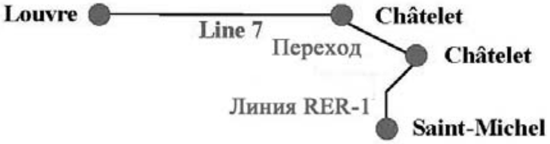


Рис. 2.9. Маршрут в метро

- Поездка по линии 7 от станции «Louvre» до станции «Châtelet» (3 остановки).
- Переход на линию RER-1.
- Поездка от станции «Châtelet» до «Saint-Michel» по линии RER-1 (1 остановка).

Этот маршрут состоит из трех этапов. Это не физический объект, который можно пнуть пяткой, как Лувр или как вашего младшего брата, но все-таки это объект.

Компоненты класса (features), команды и запросы[1]

Что делает объект объектом – не то, что он может иметь физического двойника, но то, что с ним можно манипулировать в нашей программе, используя множество хорошо определенных операций, называемых **методами (features, routines)**.

Некоторые из методов, применимых к объекту «маршрут», включают вопросы, которые мы можем задавать, например:

- Какова начальная точкой маршрута? Какова конечная точка? (В нашем примере для маршрута *Route1* таковыми являются Louvre и Saint-Michel)
- Как передвигаемся на маршруте: пешком, на автобусе, на автомобиле, метро или маршрут смешанный? (В примере – метро)
- Сколько этапов включает маршрут? (В примере – три)
- Какие линии метро используются, если они есть в маршруте? (В примере – линии 7 и RER-1.)

Методы, позволяющие получать свойства объекта, называются **запросами (queries)**.

Есть методы другого вида; они называются **командами (commands)**. Команды позволяют изменять свойства некоторых объектов. Мы уже использовали команды: в нашей первой программе *Paris*.*display* изменяет образ, показываемый на экране. Фактически, все четыре метода, вызываемые в нашей первой программе, являются командами. Вот примеры еще нескольких команд, допустимых для маршрутов.

- Удалить (Remove) – первый этап маршрута или последний.
- Присоединить (Append) (добавить в конец) – новый этап, имеющий начало в конечной точке маршрута. В нашем примере с маршрутом *Route1* новый этап может идти от станции «Сен-Мишель» до станции «Порт-Рояль». В этом случае ответы на запросы изменятся, поскольку маршрут будет включать 4 этапа и 3 линии метро.
- Добавить в начало (Prepend) новый этап, имеющий окончание в начальной точке маршрута. В нашем примере с маршрутом *Route1* новый этап может идти от станции «Опера» до станции «Лувр». Число станций при этом изменится, но число линий останется прежним, поскольку станция «*Opera*» расположена на линии 7.

Все эти операции изменяют маршрут и, следовательно, являются командами.

Мы можем, кстати, точно определить смысл высказывания, что команда «*изменяет*» объект: она изменяет видимые свойства объекта. Видимые свойства – это те свойства, которые доступны через запросы.

Например, если вы спросите о числе этапов на маршруте (запрос), затем присоедините этап (команда) и затем снова выполните запрос, то новый ответ будет на единицу больше предыдущего. Если же выполните команду «Remove», запрос после этого вернет число на единицу меньше, чем до выполнения команды.

[1] О переводе терминов. В языке Eiifel для составляющих класса используется термин «feature», который переводится как «компонент класса» или просто «компонент». Компонентами могут быть как данные класса, так и операции над ними. В Eiffel для данных используется термин «attribute», который и переводится, как «атрибут». Для операций применяется термин «feature» либо «routine». При переводе также сохраняется общий термин «компонент» либо «метод», когда из контекста ясно, как в данном разделе, что речь идет об операциях.

Знаки на въезде и выезде из туннеля в Германии на автобане являются хорошей иллюстрацией различия между командами и запросами. Знак на въезде в туннель выглядит так:

Рис. 2.10. Команда на въезде в туннель

«Licht!», говорят вам. Включите ваши фары! Безошибочно воспринимается как команда. На выезде тон изменяется на более мягкий:

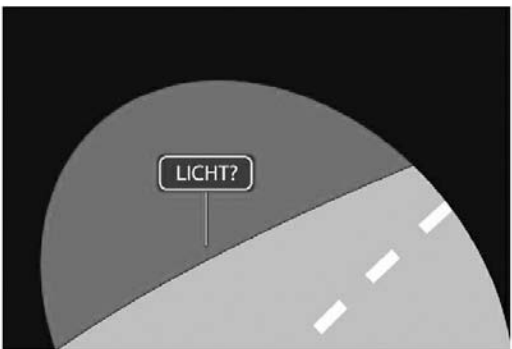


Рис. 2.11. Команда на выезде из туннеля

«Licht?»: не забыли выключить фары? Просто запрос.

Этот запрос является прекрасным примером проектирования «интерфейса пользователя», результатом тщательных исследований и стремлением избежать распространенных ошибок. Такой же подход должен быть и при проектировании интерфейса в программных системах. В частности, что было бы, если бы знак на выезде был командой «Выключите ваши фары», которой дисциплинированные водители должны были бы подчиниться даже в ночное время? Запрос же не заставляет действовать, являясь полезным напоминанием.

Объекты как машины

Первое, что мы узнали о программах, — что они являются машинами. Подобно любой сложной машине, программа во время выполнения сделана из многих маленьких машин. Наши объекты являются такими машинами.

Вероятно, трудно представить: как можно маршрут в метро представить в виде *машины*? Но фактически ответ известен: наши машины характеризуются множеством операций – командами и запросами, доступными для пользователей машины. Вспомните DVD-плейер с такими командами, как «включить», «переключить дорожку», «выключить» и с запросом «число проигранных дорожек». В нашей программе объект *Route1* в точности подобен DVD-плейеру: машине с командами и запросами.

Рисунок напоминает об этом соответствии: прямоугольные кнопки слева представляют команды; эллиптические кнопки справа являются запросами.

Рис. 2.12. Объект «route», изображенный в виде машины

Объекты, такие как *Route1,* можно рассматривать с двух точек зрения.

1. Объект представляет некоторую коллекцию данных в памяти, описывающих, как в случае с маршрутом (route), всю информацию, характеризующую объект, – число этапов, где он начинается, где заканчивается, и так далее.
2. Объект является машиной, обеспечивающей выполнение команд и запросов.

Эти две точки зрения не являются противоречащими, они дополняют друг друга. Операциям, которые машина выполняет, доступны данные объекта. Операции могут модифицировать данные.

Объекты: определение

Обобщая дискуссию об объектах, дадим точное определение, которое будет служить нам на протяжении всей книги.

Определение: объект

Объект – это программная машина, позволяющая получать доступ и модифицировать коллекцию данных.

В этом определении и в остальной части обсуждения под «получением доступа» понимается возможность получать ответы на вопросы относительно данных без их модификации (мы могли бы также говорить «запрашивать данные»).

Слова «доступ» и «модификация» отражают уже отмеченное различие между двумя фундаментальными видами операций.

Определения: метод, запрос, команда

Операция, которую программа может применять к объекту, называется **методом**, и:

- метод, получающий доступ к объекту, называется **запросом**;
- метод, который может изменить объект, называется **командой**.

Примерами команд были *display* для объекта *Paris* и *spotlight* — для *Louvre*. Запросы также рассматривались, хотя еще не встречались в тексте программ.

Запросы и команды работают на существующих объектах. Это означает, что нам нужен третий вид операций: операции создания, первоначально дающие нам объекты. На данный момент беспокоиться не стоит, поскольку все объекты, необходимые в этой главе, уже созданы – *Paris*, *Louvre*, *Route1* ... – как часть «магии» класса *TOURISM*. Во время выполнения ваша программа сможет использовать их. Вскоре (в главе 6) вы научитесь создавать свои собственные объекты.

Там же будет объяснено, почему понятие «машина» относится не только к объектам, но и к классам.

2.4. Методы с аргументами

Запросы так же важны, как и команды. Приведем теперь несколько примеров их использования. Мы можем, например, захотеть узнать начальную точку нашего маршрута. Для этого воспользуемся запросом *origin* (начало). Напишем:

```
Route1.origin
```

Это вызов метода, подобно вызову команд *Route1.animate* и других. В данном случае, поскольку метод является запросом, он ничего не делает, он просто вырабатывает значение — начало маршрута *Route1.* Мы можем использовать это значение разными способами, например, напечатать на листе бумаги, но позвольте нам отобразить его на экране.

Вы, наверное, заметили, рассматривая отображение программной системы в среде EiffelStudio, в нижней части дисплея маленькое окошко (прямоугольную область), помеченное как «Console» и используемое для показа информации о состоянии системы, которая моделирует город. В нашей программе *Console* (отгадайте с первого раза, что это?) – объект. Он является одним из предопределенных объектов, подобно *Paris* и *Route1*, которые наш класс *PREVIEW наследует* от *TOURISM*.

Одна из команд, применимых к *Console,* называется *show*; результатом ее выполнения является выдача некоторого текста на консоль. Сейчас мы воспользуемся ею для показа начальной точки маршрута.

Время программировать!

Отображение специфической информации

Модифицируем теперь нашу предыдущую программу, чтобы она дополнительно позволяла отобразить информацию о начальной точке маршрута.

Необходимы только два изменения: обновить комментарий — в интересах пояснения текста – и добавить в конец новую операцию:

```
class PREVIEW inherit
    TOURISM
feature
    explore
            -- Показать информацию о городе? маршруте и начале маршрута
        do
            Paris.display
            Louvre.spotlight
            Line8.highlight
            Route1.animate
            Console.show (Route1.origin)
        end
end
```

Выполните полученную программу. Началом маршрута является Лувр (формально: *Palais Royal Musee du Louvre*), что показано в окне консоли:

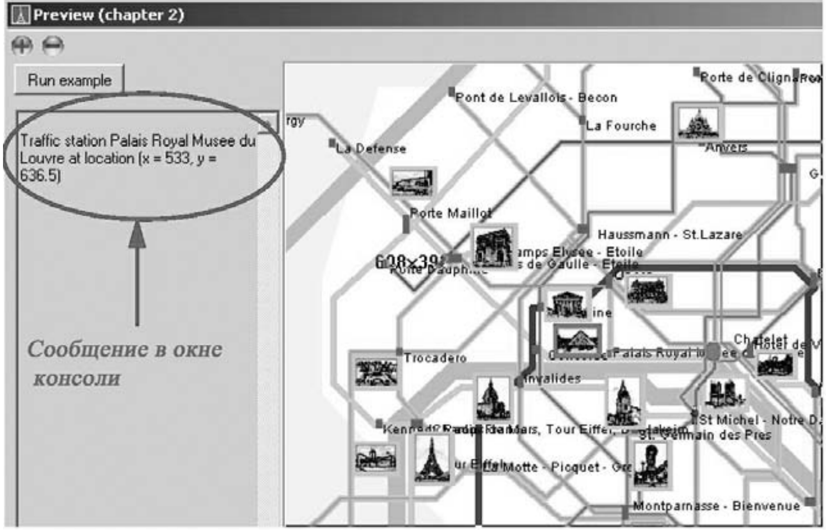


Рис. 2.13. Объект «route», изображенный в виде машины

Это результат вызова нового метода *Console.show* (*Route1.origin*). Все предыдущие вызовы *имели форму some_object.some_feature*, но для этого вызова форма изменилась:

```
some_object.some_feature (some_argument)
```

где *some_argument* является значением, которое мы передаем методу, поскольку оно необходимо для выполнения работы. Методу *show* необходимо знать, что же он должен показать, — мы и передаем ему соответствующее значение.

Такие значения известны как аргументы метода, аналогично аргументам функций в математике, где *cos* (*x*) обозначает косинус от *x* – функцию *cos,* примененную к аргументу *x*.

Некоторые методы могут иметь несколько аргументов (разделенных символом «запятая»). В хорошо спроектированном ПО у большинства методов аргументов мало, обычно ноль или один аргумент.

Понятие аргумент дополняет наш багаж, состоящий из базисных программных элементов, которые служат основой всех обсуждений в последующих главах: классы, компоненты, методы, аргументы, команды, запросы и объекты.

2.5. Ключевые концепции, изученные в этой главе

- Программную систему можно представлять как множество механизмов, позволяющих создавать, получать доступ и изменять коллекции информации, называемых *объектами*.
- Объект – это машина, которая управляет некоторой коллекцией данных, предоставляемых программой в момент выполнения, с множеством операций (они называются *методами*), применимых к этим данным.
- Методы бывают двух видов: запросы, которые возвращают информацию об объекте, и команды, которые могут изменять объект. Применение команд к объекту может изменить результаты последующих запросов.
- Некоторые объекты являются программными моделями *объектов* физического мира, другие – программными моделями *концепций* физического мира, подобно маршруту путешествия, третьи могут *не иметь аналогов* в физическом мире, имея смысл только в рамках самого ПО.
- Базисными операциями, выполняемыми программами, являются *вызовы методов*, каждый из вызовов применяет некоторый метод к некоторому целевому объекту.
- Метод может иметь *аргументы*, предоставляющие необходимую методу информацию.

Новый словарь

Argument	Аргумент	**Class**	Класс
Command	Команда	**Declaration**	Объявление
Feature	Метод	**Feature call**	Вызов метода
Indentation	Отступ	**Object**	Объект
Query	Запрос		

Более полное определение класса будет дано в главе 4.

2-У. Упражнения

2-У.1. Словарь

Дайте точное определение всех элементов словаря.

2-У.2. Карта концепций

Это упражнение необходимо будет выполнять на протяжении всех последующих глав, расширяя полученные здесь результаты. Цель состоит в создании карты терминов, вводимых в словарях, которые сопровождают каждую главу. Разместите термины в квадратиках на листе бумаги и свяжите их стрелками, отражающими отношение между понятиями. Дайте имена каждому из отношений.

> Конечно, можно использовать и электронные средства для построения диаграммы, но одного листа бумаги достаточно, чтобы отобразить связи понятий данной главы.

Можно использовать различные имена отношений, но в любом случае рассмотрите следующие фундаментальные отношения.

- «Является специальным подвидом». Например, в области знаний, не относящейся к программированию, «*журнал*» является подвидом «*публикации*». Другим подвидом «*публикации*» является «*книга*».
- «Является экземпляром». «*Австралия*» является экземпляром «*страны*». Не путайте с предыдущим отношением. Австралия не является подвидом страны — это страна.
- «Основано на» (применимое к концепциям). Например, в математике «деление» основано на «умножении», так как невозможно понять деление, не поняв ранее суть умножения. Если одно из двух предыдущих отношений существует для двух терминов и для них также имеется и отношение «основано на», то следует применять это отношение лишь тогда, когда один из терминов не является ни подвидом, ни экземпляром другого термина.
- «Содержит». Например, каждая *страна* содержит *город*. Вместо отношения «содержит» можно применять инверсное к нему – «является частью».

Наряду с этими отношениями можно применять и собственные отношения, при условии, что вы дадите им точные определения.

2-У.3. Команды и запросы

Представьте, что вы создаете ПО для работы с документами – создания, модифицирования и доступа к ним. Предположите, что вы проектируете класс *WORD* («Слово»), который описывает понятие «слово», и класс *PARAGRAPH* («Абзац»), описывающий понятие абзаца. Для каждого из следующих возможных методов класса *PARAGRAPH* установите, какой из них должен быть командой, а какой – запросом.

1. Метод *word_count*, используемый в вызовах *my_paragraph*.*word_count*, возвращающий число слов абзаца.
2. Метод *remove_last_word*, используемый как *my_paragraph*.*remove_last_word*, удаляющий последнее слово абзаца.
3. Метод *justify*, используемый как *my_paragraph*.*justify*, позволяющий убедиться, что абзац выровнен в соответствии с установленными границами для левого и правого поля.

4. Метод *extend*, используемый как *my_paragraph*.*extend* (*my_word*), имеющий слово в качестве аргумента и добавляющий его в конец абзаца.
5. Метод *word_length*, используемый как *my_paragraph*.*word_length* (*i*), у которого целочисленный аргумент задает индекс (порядковый номер) слова в абзаце, а в качестве результата возвращается число символов этого слова.

2-У.4. Проектирование интерфейса

Представьте, что вы создаете ПО для работы с MP3-плейером.

1. Перечислите основные классы, которые вы будете использовать.
2. Для каждого такого класса перечислите применимые методы, указав, будет ли метод командой или запросом, будет ли он иметь аргументы и каков их смысл, если они есть.

3

Основы структуры программ

Предыдущая глава позволила нам получить первое представление о программах. Теперь мы готовы к введению новых концепций. Давайте ближе познакомимся с некоторыми частями программы, уже использованными, но не получившими пока собственные имена.

3.1. Операторы (instructions) и выражения

Основными операциями предыдущей программы, заставляющие компьютер выполнять определенные действия, были:

```
Paris.display
Louvre.spotlight
Line8.highlight
Route1.animate
Console.show (Route1.origin)
```

Такие операции в языке программирования естественно называются операторами языка. Обычно принять записывать в каждой строке программы по одному оператору, что облегчает читабельность наших программ.

До сих пор все рассматриваемые нами операторы были операторами вызова метода. В последующих главах мы встретимся с другими видами операторов языка.

Для выполнения своей работы операторам нужны некоторые значения, подобно тому, как математическая функция «косинус» при ее вызове $cos(x)$ может дать результат, только если известно значение x. При вызове метода необходимыми значениями являются:

- *цель*, заданная объектом. В наших примерах это *Paris*, *Louvre* и т. д.;
- *аргументы*, не обязательные, но необходимые для некоторых вызовов, как *Route1.origin* в последнем примере.

Такие элементы программы, обозначающие значения, называются *выражениями*. Наряду с продемонстрированной формой записи выражения нам встретятся выражения в привычной математической форме, такие как $a + b$.

Определение: операторы, выражения

В тексте программ:

- оператор обозначает базисную операцию, которую необходимо выполнить в период работы программы;
- выражение обозначает значение, используемое оператором при его выполнении.

3.2. Синтаксис и семантика

В определении оператора и выражения важную роль играет слово «обозначает». Выражение, такое как *Route1.origin* или $a + b$, не является значением – это последовательность слов программного текста. Оно обозначает значение, которое может существовать в момент выполнения.

Аналогично оператор, такой как *Paris.display*, является некоторой последовательностью слов, скомбинированной в соответствии с некоторыми структурными правилами; он *обозначает* некоторую операцию, которая будет происходить в момент выполнения.

Этот термин «*обозначает*» отражает различие между дополняющими друг друга аспектами программ:

- способа записи программы, состоящей из слов, которые, в свою очередь, составлены из символов, печатаемых на клавиатуре. Например, оператор *Paris.display* состоит из трех частей – слова, составленного из пяти символов *P*, *a*, *r*, *i*, *s*, затем «точки», затем слова, составленного из семи символов.
- эффекта от этих элементов программы, который, как вы ожидаете, возникнет в процессе выполнения: вызов метода *Paris.display* приведет к отображению на экране карты Парижа.

Первый аспект характеризует *синтаксис* программы, второй – ее *семантику*. Дадим точные определения.

Определения: синтаксис, семантика

Синтаксис программы – это структура и форма записи ее текста.
Семантика – множество свойств потенциально возможных выполнений программы.

Так как программы пишутся для того, чтобы их можно было выполнять и получать результат этой работы, определяющим фактором является семантика, но без синтаксиса не было бы правильного текста, следовательно, не было бы и выполнения, не имела бы значения семантика. Так что обоим аспектам программы следует уделить должное внимание.

Ранее у нас уже было разделение: команды против запросов. Команды являются *императивными*: они командуют, заставляя компьютер при запуске программы выполнять некоторые действия, которые могут изменять объекты. Запросы являются *дескриптивными*: они запрашивают у компьютера некоторую информацию об объектах без изменения самих объектов. Эта информация предоставляется программе. Комбинируя эти различия с различиями в синтаксисе и семантике, приходим к четырем различным ситуациям.

	Синтаксис	**Семантика**
Императивный	Оператор	Команда
Дескриптивный	Выражение	Запрос
		Значение

В нижнем правом углу таблицы имеем две семантики: *запрос* является программным механизмом для получения некоторой информации; эта информация, полученная при выполнении запроса, создана из *значений*.

3.3. Языки программирования, естественные языки

Нотация, определяющая синтаксис и семантику программ, называется **языком программирования**. Существует множество языков программирования, служащих различным целям. Язык, использованный в этой книге, называется Eiffel (Эйфель).

Языки программирования являются искусственными творениями. Называя их языками, мы предполагаем возможность их сравнения с естественными языками, подобных английскому или русскому. Языки программирования обладают некоторыми общими чертами со своими двоюродными братьями.

- Общая организация текста в виде последовательности слов, состоящих из символов. Точка есть точка, как в Eiffel, так и в английском и в русском языках.
- И в естественных, и в искусственных языках синтаксис языка, определяющий структуру текста, отличается от семантики, определяющей смысл.
- Доступны слова с предопределенным смыслом, такие, как «the» в английском и «do» в Eiffel. Наряду с этим есть возможность определять собственные слова, как это делал Льюис Кэролл в «Алисе в Зазеркалье»: «Варкалось. Хливкие шорьки пырялись по наве...», так же, как это делали мы, назвав наш первый класс *PREVIEW* именем, не имеющим специального смысла в Eiffel.

Вполне возможно назвать класс именем «Шорек», а его метод – «Варкалось». Правда, такие имена идут вразрез с правилами стиля.

Создание имен – это общая, без конца возникающая необходимость в языках программирования. В естественных языках вам не приходится изобретать новые слова, если только вы не поэт, не ребенок и не ботаник, специализирующийся на флоре бассейна Амазонки. Программист, внешне выглядящий вполне взрослым и не имеющий никакого отношения к цветкам из Амазонки, может в течение дня придумать несколько десятков новых имен.

- Eiffel усиливает аромат естественного языка, преобразуя слова английского языка в свои ключевые слова. Каждое ключевое слово Eiffel является одним общеиспользуемым словом английского языка.

Некоторые языки программирования предпочитают использовать аббревиатуры, такие как *int* вместо *INTEGER.* Мы предпочитаем для ясности использовать полные слова. Единственным исключением является термин *elseif*, составленный из двух слов.

- Рекомендуется всюду, где это возможно, использовать слова из английского или родного языка с содержательным смыслом для определяемых вами имен, как это делали мы в построенных до сего момента примерах: *PREVIEW*, *display*, или *Route1.*

Схожесть языков программирования и естественных языков полезна, поскольку способствует пониманию программ. Но не стоит заблуждаться – языки программирования отличаются от естественных языков. Они являются *искусственными* нотациями, спроектированными для решения специфических задач. В этом их сила и слабость.

- Выразительная сила языков программирования неизмеримо мала в сравнении с любым естественным языком даже на уровне ребенка четырех лет. Языки программирования не позволяют выражать чувства или мысли. Они лишь могут позволить определить объекты, представимые в компьютере, и задачи, выполнимые над этими объектами.

- Проигрывая в выразительности, языки программирования выигрывают в точности. Текст на естественном языке грешит двусмысленностями и открыт для множества интерпретаций, что придает ему некоторое обаяние. Когда же мы общаемся с компьютером, задавая программу действий, нельзя позволить себе приблизительности, мы ожидаем строго определенных результатов. Синтаксис и семантика языка программирования должны быть определены совершенно строго.

Почувствуй стиль

Естественные языки в ваших программах

Естественному языку отводится важная роль в программах: в комментариях. Мы уже говорили, что любой программный текст, начинающийся с двух тире «—» и вплоть до конца строки, является комментарием. В отличие от остального программного текста, комментарии не следуют точным правилам синтаксиса, поскольку они не участвуют в выполнении программы и, следовательно, не имеют семантики. Комментарии объясняют программу, помогая людям лучше понять ее смысл.

Естественные языки служат основой построения идентификаторов, в частности, имен классов и методов. Методологический совет: используйте полные, содержательные имена, такие как *METRO_LINE* для имени класса. Аббревиатуры следует применять только в тех случаях, если они используются в естественном языке.

Называя наши нотации «языками», мы оказываем им не вполне заслуженную честь – они скорее представляют расширенную версию *математической нотации*, используемой для записи формул.

Термин «код», означающий текст в языке программирования, отражает этот факт. Он используется в выражении «Строка кода», как в тексте «*Windows Vista* содержит более 50 миллионов строк кода». Мы иногда называем процесс программирования кодированием, подчеркивая этим немного уничижительным термином, что речь идет о процессе низкого уровня, исключающего проблемы проектирования, как, например, в данной фразе: «Они думают, что все идеи уже высказаны, и все, что остается, – это простое кодирование». «Кодировщик» – уничижительное название программиста.

И все же, языки программирования имеют собственную красоту, которую, я надеюсь, вы сумеете ощутить в процессе обучения. Если вы начнете думать о ваших отношениях с любимой в терминах *relationship.is_durable* (отношение.является_прочным) или пошлете родителям SMS: *Me.account.wire* (*month.allowance* + (*month*+1).*allowance* + 1500, *Immediately*) (Мне.счет. телеграфируй (месяц.карманные_расходы + (месяц +1).карманные_расходы + 1500, срочно), то это значит:

1) вы твердо усвоили концепции;
2) следует отбросить эту книгу и отдохнуть пару недель.

3.4. Грамматика, категории, образцы

Для описания синтаксиса естественного языка – структуры текстов – лингвисты задают грамматику языка. Для простого предложения

Анна звонит друзьям

типичная грамматика скажет нам, что это частный случай (мы говорим «образец») некоторой «категории», называемой в грамматике: «Простое_глагольное_предложение». Предложение состоит из трех компонентов, каждый из которых является образцом некоторой категории:

- субъект действия: *Анна*, образец категории «Существительное»;
- действие, описываемое в предложении: *звонит* – образец категории «Глагол»;
- объект действия: *друзьям* – еще один образец «Существительного».

Ровно эти же концепции будут применяться для описания синтаксиса языков программирования. Например:

- категорией в грамматике Eiffel является Class, описывающий все программные тексты классов, которые кто-либо может написать;
- частные случаи текста классов, как класс *PREVIEW* или класс *TOURISM*, являются *образцами* категории Class.

В главе 11 мы в деталях опишем синтаксис языка. Сейчас же достаточно ограничиться определением.

Определения: грамматика, категория, образец

Грамматика языка программирования – это описание его синтаксиса.
Категория (грамматическая) – элемент грамматики, описывающий некоторую категорию возможных синтаксических элементов в соответствующем языке.
Образец категории – это синтаксический элемент.

Отметим соотношение между категорией и образцом. Категория задает тип синтаксического элемента, образец является экземпляром этого типа. Так:

- в обычной грамматике мы можем иметь грамматические категории существительного и глагола. «*Анна*» является образцом существительного, «звонит» – образец глагола;
- в грамматике Eiifel мы можем иметь категории: Class и Feature. Любой текст класса является образцом категории Class, текст метода является образцом категории Feature.

Для имен, задающих категории элементов, будем использовать специальный стиль, чтобы отличать их от элементов программы.

3.5. Вложенность и структура синтаксиса

Синтаксическая структура программного текста допускает несколько уровней образцов. Класс является образцом, это же верно и для части класса – оператора, и для имени метода. Языки программирования поддерживают возможность встраивания одних образцов в другие, технический термин – **вложенность** (**nesting**). Например, класс может содержать несколько методов, методы могут содержать операторы, те, в свою очередь, содержат выражения. Вот пример вложенной структуры образцов для ранее приведенного класса. Для простоты оставлены только два оператора и класс назван *PREVIEW1*, чтобы отличать его от полной версии:

Встроенные прямоугольники подсвечивают вложенные образцы. Самый внешний прямоугольник задает объявление класса, содержащее наряду с другими образцами объявление метода, частью которого является тело метода, содержащее два оператора, и так далее.

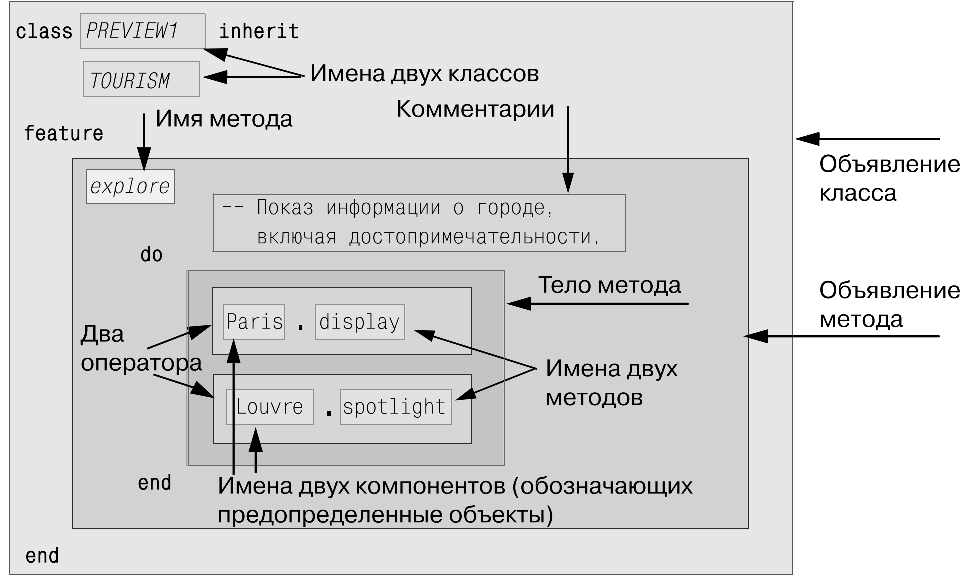


Рис. 3.1. Пример синтаксической структуры

Некоторые элементы синтаксиса, такие как точка и ключевые слова **class**, **do**, **end**, играют роль разделителей и сами по себе не несут семантического смысла. Мы не рассматриваем их как образцы.

Убедитесь, что вы понимаете представленную здесь синтаксическую структуру.

3.6. Абстрактные синтаксические деревья

Для больших программных текстов более подходит другая структура, отражающая синтаксис. Она основана на понятии «дерева», часто используемого для отображения организационной структуры компании. Это понятие ассоциируется с настоящими деревьями, их ветвями и листьями. Правда, у наших деревьев, в отличие от настоящих, корень дерева располагается вверху и дерево «растет» сверху вниз или слева направо. Дерево имеет корень, ветви которого идут к другим узлам, а те, в свою очередь, могут иметь ветви, или быть листом дерева. Деревья служат для представления иерархических структур, как показано ниже:

Рисунок представляет **абстрактное синтаксическое дерево**. Оно абстрактно, поскольку не включает элементы, играющие роль разделителей, такие как **do** и **end**. Мы можем построить конкретное синтаксическое дерево, включающее такие элементы.

Дерево включает узлы и ветви (вершины и дуги). Каждая ветвь связывает один узел с другим. У данного узла может быть несколько выходящих ветвей, но в каждый узел ведет максимум одна ветвь. Узел, у которого нет входящей в него ветви, называется **корнем**. Узлы без выходящих ветвей называются **листьями**. Узел, не являющийся ни корнем, ни листом, называется **внутренним узлом**.

Непустое дерево имеет в точности один корень (структура, представленная нулем, одним или несколькими раздельными деревьями, имеющая произвольное число корней, называется *лесом*).

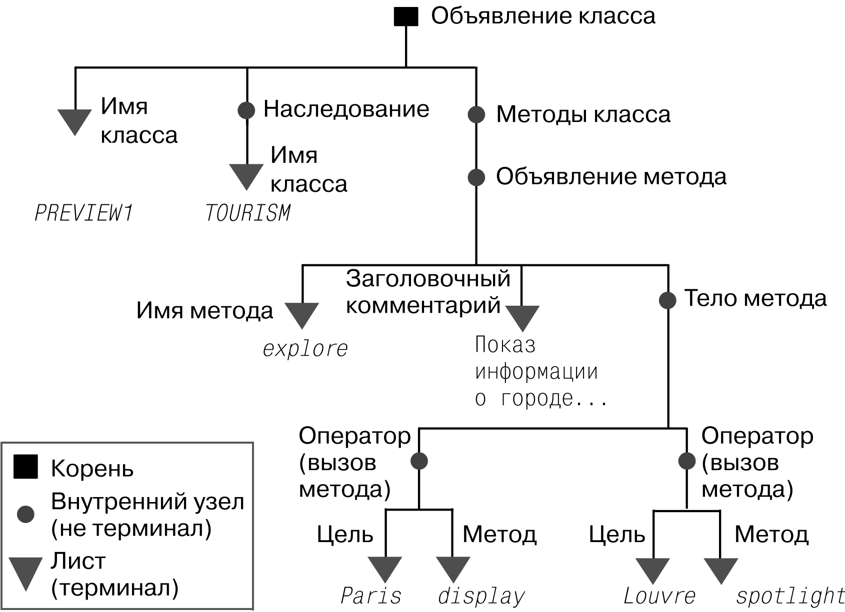


Рис. 3.2. Абстрактное синтаксическое дерево

Деревья являются важными структурами в информатике, нам часто придется с ними встречаться в разных контекстах. Здесь мы рассматриваем их как деревья, задающие структуру синтаксиса элемента программы – класса. Она представлена вложенными образцами с тремя видами узлов.

- Корень, представляющий общую структуру, – самый внешний прямоугольник на предыдущем рисунке.
- Внутренние узлы, которые представляют подструктуры, содержащие вложенные образцы – например, вызов метода содержит цель и имя метода.
- Листья, представляющие образцы, которые не содержат последующих вложений, такие как имена классов и методов.

Листья абстрактного синтаксического дерева называются также **терминалами**, а корень и внутренние узлы – **нетерминалами**.

Каждый образец относится к специальному виду: верхний узел задает класс, другие представляют имя класса, предложение «наследования», множество объявлений методов. Каждый такой вид образца является грамматической категорией. На рисунке дерева для каждого узла приведено имя категории. Категория может быть как терминальной, так и нетерминальной, что зависит от образцов, представляющих категорию. Рисунок показывает, что «Объявление метода» является нетерминальной категорией, а имя метода – терминальной.

Категория определяет общее синтаксическое понятие. Синтаксис языка программирования определяется множеством категорий и их структурой.

3.7. Лексемы и лексическая структура

Базисными составляющими синтаксической структуры являются терминалы, ключевые слова, специальные символы, такие как точка в вызове метода. Эти базисные элементы называются **лексемами (tokens)**.

> Лексемы подобны словам и символам обычного языка. Предложение «Ах, ох – это восклицания!» содержит четыре слова и три символа.

Виды лексем

Лексемы бывают двух видов.

- **Терминалы** соответствуют, как мы видели, листьям абстрактного синтаксического дерева, каждое из которых несет некоторую семантическую информацию. Они включают имена, такие как *Paris* или *display*, называемые идентификаторами и выбираемые каждым программистом для именования семантических элементов, например, объектов (*Paris*) и методов (*display*). Другими примерами являются знаки операций, такие как + и <=, появляющиеся в выражениях, и литеральные константы, обозначающие значения с самообъявлением, например, целое 34. Синтаксис литеральных констант строится так, чтобы по их записи однозначно определялся их тип.
- **Ограничители** играют чисто синтаксическую роль и не несут никакой семантики. Они включают 65 ключевых слов, таких как **class**, **inherit**, **feature**, и специальные символы, например, точку и двоеточие. Они не появляются в абстрактном синтаксическом дереве, но появятся как листья при построении конкретного синтаксического дерева.

Уровни описания языка

Форма лексем определяет **лексическую** структуру языка. Синтаксический уровень является надстройкой – более высоким уровнем по отношению к лексическому, а семантический уровень выше синтаксического.

- Лексические правила определяют, как создаются лексемы из символов.
- Синтаксические правила определяют, как создаются образцы из лексем, удовлетворяющих лексическим правилам.
- Семантические правила определяют эффект программ, удовлетворяющих синтаксическим правилам.

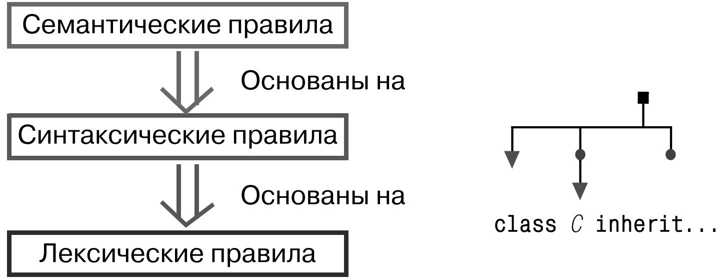


Рис. 3.3. Уровни описания языка

Важной особенностью этой иерархии является то, что свойства любого уровня определены только при условии выполнения ограничений на предыдущих уровнях. Синтаксис определен только для лексически корректных текстов, а семантика существует только для синтаксически корректных программ.

Мы встретимся с еще одним специальным уровнем, лежащим между синтаксисом и семантикой: *обоснованием* (*validity*), на котором рассматриваются правила, не относящиеся к синтаксису, например, ограничения типа.

Идентификаторы

На данный момент нам необходимо только одно лексическое правило для идентификаторов.

Синтаксис: идентификаторы

Идентификатор начинается с буквы, за которой следует ноль или более символов, каждый из которых может быть:

- буквой;
- цифрой (0-9);
- символом подчеркивания «_».

Примером идентификатора, содержащего цифры, является *Route1*.

Оставаясь в рамках этого правила, вы можете создавать собственные идентификаторы. Действует одно исключение: нельзя использовать в роли идентификаторов ключевые слова, зарезервированные для специальных целей (на данный момент нам известно лишь несколько ключевых слов, но, если ошибочно мы попытаемся использовать в роли идентификатора ключевое слово, появится сообщение об ошибке).

Почувствуй стиль

Выбирая идентификаторы

Для повышения читабельности программ выбирайте идентификаторы, четко идентифицирующие предполагаемую роль, за исключением специальных случаев, которые вскоре рассмотрим. Используйте полные имена, а не аббревиатуры: *Route1*, а не *R1* или *Rte1*.

Нет налога на нажатие клавиш. Те несколько секунд, которые вы, возможно, сэкономите, опустив несколько символов, ничего не стоят в сравнении с тем временем, когда вам или кому-либо другому понадобится понять, что же делается в вашей программе.

В идентификаторах, обозначающих сложные понятия, используйте подчеркивание для разделения последовательности слов, как в *My_route* или *bus_station*. Это относится и к именам классов, записываемых в верхнем регистре: *PUBLIC_TRANSPORT*. Не переусердствуйте: для большинства идентификаторов достаточно одного слова или двух слов, соединенных подчеркиванием. Ясность не означает многословие.

В некоторых программах, хотя для программ на Eiffel это не так, можно встретиться с многословными идентификаторами, в которых каждое следующее слово начинается с заглавной буквы: myRoute, PublicTransport. Это соглашение называется верблюжьим стилем (camel style) из-за возникающего горба. Лучше не следовать этому стилю, поскольку «онНамногоХужеЧитается», чем стиль с подчеркиванием, «остающийся_совершенно_ясным_даже_для _длинных_идентификаторов».

Белые пробелы и отступы

Лексическая структура состоит из последовательности лексем. Для разделения соседствующих лексем можно использовать «белый пробел» (**break**), который может быть:

- пробелом;
- символом табуляции (задающим последовательность пробелов, которая позволяет перейти к фиксированной позиции в строке);
- символом перехода на новую строку.

Белые пробелы служат только для разделения лексем. Для синтаксиса и семантики нет разницы, как вы разделяете лексемы. Разрешается использовать один или несколько пробелов, переходы на новую строку или табуляцию. Такая гибкая структура, известная как «*свободный формат*», позволяет вам создать раскладку текста программы, отражающую структуру программы, что улучшает читабельность программы, как это сделано в примерах нашей книги.

Ваша программа хранится в файле, который содержит последовательность символов, таких как буквы, цифры, знаки табуляции и другие специальные символы. Для большинства используемых сегодня файловых форматов для перехода на новую строку применяется один специальный символ «новая строка». Возможно, вы встречались и с символом «перевода каретки» (*Carriage Return*). Это название пришло из прошлого, когда тексты печатались на пишущей машинке, каретка смещалась в процессе печати строки и для перехода на новую строку необходимо было предварительно вернуть каретку в начальную позицию. В операционной системе Windows переход на новую строку кодируется последовательностью двух символов – возврата каретки и новой строки.

Белый пробел обычно не требуется между идентификатором и символом: можно писать $a+b$ без всяких пробелов, поскольку не возникает двусмысленности. Однако правила стиля требуют использования белых пробелов для улучшения ясности текста: $a + b$.

3.8. Ключевые концепции, изученные в этой главе

- Для записи программ используются языки программирования.
- Программы имеют лексическую структуру, определяющую форму базисных элементов. Лексемы разделяются «белыми» пробелами – табуляцией, возвратом строки, пробелом.
- Программы имеют синтаксическую структуру, которая определяет иерархическую декомпозицию на элементы (образцы), построенные из лексем.
- Программы имеют семантику, определяющую эффект времени выполнения каждого образца и всей программы в целом.
- Синтаксическая структура обычно включает вложенность и может быть задана в виде дерева, известного как *абстрактное синтаксическое дерево*.

Новый словарь

Abstract syntax tree (AST)	Абстрактное синтаксическое дерево (АСД)	**Break**	Белый пробел
Code	Код	**Carriage return**	Возврат каретки
Delimiter	Ограничитель	**Construct**	Категория
Free format language	Свободный формат языка	**Expression**	Выражение
Instruction	Оператор	**Grammar**	Грамматика
Leaf	Лист	**Identifier**	Идентификатор
Natural language	Естественный язык	**Internal node**	Внутренний узел
New line	Новая строка	**Lexical**	Лексический
Nonterminal	Нетерминал	**Nesting**	Вложенность
Root	Корень	**Node**	Узел
Special symbol	Специальный символ	**Operator**	Знак операции
Syntax	Синтаксис	**Semantics**	Семантика
Token	Лексема	**Specimen**	Образец
Value	Значение	**Terminal**	Терминал
		Tree	Дерево

3-У. Упражнения

3-У.1. Словарь

Дайте точные определения терминам словаря.

3-У.2. Карта концепций

Добавьте новые термины в карту концепций, спроектированную в предыдущей главе.

3-У.3. Синтаксис и семантика

Для каждого из следующих предложений укажите, характеризуют ли они синтаксис, семантику, то и другое, ни то, ни другое (объясните решение).

1. При вызове метода целевой объект должен отделяться точкой от имени метода.
2. При вызове метода *x.f* нет необходимости помещать пробелы перед или после точки, хотя они являются допустимыми.
3. Каждый вызов метода применяет метод к определенному объекту – цели вызова.
4. Если у метода есть аргументы, то они должны заключаться в круглые скобки.
5. Два или более аргумента заданного типа разделяются запятыми.
6. Операторы, разделенные символом «точка с запятой», будут выполняться один после другого.
7. Eiffel и Smalltalk являются объектно-ориентированными языками.

4

Интерфейс класса

В предыдущих главах мы начали строить некоторое ПО, основываясь на существующих элементах. Теперь мы собираемся сделать нечто большее, рассмотрев, как можно использовать ранее написанные классы. Это даст возможность по-новому взглянуть на понятие класс – фундамент всего дальнейшего программирования. Мы введем в рассмотрение новые концепции – *интерфейса* и *контракта*.

Система, содержащая примеры этой главы, находится в подкаталоге *04_interface* каталога *examples* ПО Traffic.

4.1. Интерфейсы

Ключевые решения, связанные с построением и использованием программных систем, включают понятие интерфейса. Определим это понятие, связывая его с понятиями «клиент» и «поставщик», имеющими самостоятельную ценность.

Определения: Клиент, Поставщик, Интерфейс

Клиентом программных механизмов (client) является система любого вида: человек; система, не являющаяся программной; элемент программной системы. Используемое клиентами ПО является для них **поставщиком** (supplier).

Интерфейс (interface) программных механизмов – это описание, позволяющее клиентам использовать механизмы поставщика.

Говоря неформально, интерфейс части ПО – это описание того, как остальной мир может «общаться» с этой частью ПО.

Определение говорит о «некотором интерфейсе» (an interface), а не о вполне определенном интерфейсе (the interface). Есть несколько различающихся видов интерфейса. Особенную важность представляют такие два вида, как:

- **интерфейс пользователя** (**user interface**), когда предполагаемыми клиентами являются люди, использующие ПО;
- **программный интерфейс** (**program interface**), предназначенный для клиентов, которые сами являются программными элементами.

В качестве примера интерфейса пользователя рассмотрим Web-браузер, частично показанный ниже. Его пользовательский интерфейс задает описание того, что могут люди делать, используя этот браузер. Интерфейс включает:

- спецификацию полей, в которых пользователи могут печатать собственные тексты, как, например, поле адреса в верхней части браузера;
- свойства кнопок («Back» и др.), которые пользователи могут нажимать для получения определенного эффекта;
- соглашения для гиперссылок (щелчок левой кнопкой мыши переводит к новой странице, правой кнопкой – задает переход к меню и т.д.).

В целом, это множество правил, покрывающее взаимодействие между браузером и его пользователями.

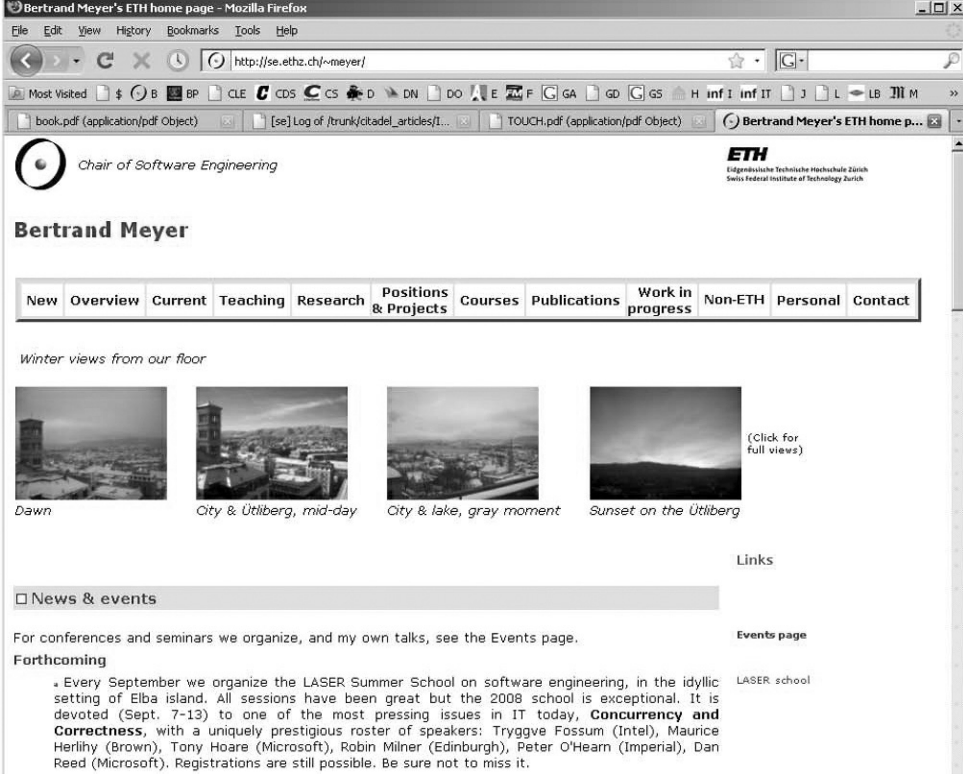


Рис. 4.1. Интерфейс пользователя

Такой пользовательский интерфейс называется графическим, поскольку он включает картинки и другие визуальные диалоговые элементы, такие как кнопки и меню. В программировании, склонном к акронимам, такой интерфейс называют **GUI** (Graphical User Interface, произносится «*Gooey*» или просто **UI** – «*You-I*»).

Другие пользовательские интерфейсы могут не включать графику, используя только текст, как в старых мобильных телефонах. Они называются текстовыми интерфейсами или интерфейсом командной строки.

Если вы имели дело с компьютерными системами, то, возможно, встречались с такими, интерфейс которых с трудом можно назвать дружественным. Вероятно, вы согласитесь, что проектирование GUI – это важная часть проектирования ПО.

Но для нашего обсуждения более важен второй вид интерфейсов – программные интерфейсы. Здесь также используется трехбуквенный акроним API (Abstract Program Interface, старая расшифровка для буквы A – «*Application*»).

В оставшейся части главы мы изучим, как выглядит API для важного частного случая программного элемента – класса. С этого момента мы сосредоточимся только на этом виде интерфейсов, не будем рассматривать интерфейс пользователя, так что для нас слова «интерфейс», API, «программный интерфейс» будут означать одно и то же понятие.

4.2. Классы

В предыдущем обсуждении мы определили объект как машину, позволяющую программе получать доступ и модифицировать коллекции данных. Коллекции данных, как показано в наших примерах, могут представлять:

- город, где операции доступа и модификации могли включать поиск характеристик транспортной системы, добавление новых транспортных средств. В качестве примера использовался город Париж, но можно получить объект, представляющий любой другой город, если обеспечить для него релевантные данные;
- маршрут путешествия. И снова можно иметь различные маршруты, а не только объект *Route1*, используемый в примерах;
- список автомобилей, остановившихся на красный свет светофора. Опять речь идет о множестве объектов;
- элементы GUI, такие как кнопки и окна на экране компьютера. Их может быть довольно много.

Внутри каждой категории объектов, например, для всех объектов, представляющих города, существует сильная схожесть. Операции, применимые к объекту *Paris*, будут также применимы к любому другому объекту – городу, скажем *New_York* или *Tokyo*. Эти операции не применимы к объектам другой категории, например, к объекту *Route1*. Аналогично операция добавления новой ветки в маршрут, допустимая для *Route1*, допустима и для всех других объектов, задающих маршрут.

Все это говорит нам, что объекты, с которыми мы манипулируем в наших программах, классифицируются естественным образом, составляя **классы**: класс, представляющий города, класс для маршрутов, класс объектов, задающих кнопки на экране, и так далее.

«Класс» на самом деле – это технический термин. Характеризует объекты данного класса общее множество применимых операций – *методов* (*features*). Отсюда следует определение:

Определение: Класс

> Класс является описанием множества объектов периода выполнения, к которым применимы одни и те же методы.

В программах имена классов всегда будем писать заглавными буквами: *CITY*, *ROUTE*, *CAR_LIST*, *WINDOW*. Имена объектов всегда будут строиться из строчных букв, но если речь идет о предопределенных объектах, таких как *Paris*, то их имена будут начинаться с заглавной буквы.

Класс задает категорию предметов, объект представляет один из этих предметов. Следующие определения задают точное отношение между классами и объектами.

Определения: Экземпляр, Генерирующий класс

Если объект *O* – один из объектов, заданных классом *C*, то *O* является **экземпляром** (instance) класса *C*, и *C* является **генерирующим классом** (generating class) для O.

Класс *CITY* представляет все возможные города (поскольку мы решили моделировать их в нашей программе). Объект *Paris* обозначает экземпляр этого класса.

Отношение между классами и объектами является типичным отношением между категорией и ее членами. «Human» – это категория, описывающая людей, «Socrates» – человек, член этой категории. Моделируя эти понятия в программной системе, мы бы создали класс *HUMAN,* и один из объектов этого класса мог быть назван *Socrates*.

Существенное различие между объектами и классами проявляется в ПО. Классы являются статическими, объекты – динамическими.

- Классы существуют только как тексты программ. Как следует из определения, класс – это описание; оно задано **текстом класса**, программным элементом, описывающим свойства ассоциированных объектов (экземпляров класса). Программа – это коллекция, состоящая из текстов классов.
- Объекты – коллекция данных – существуют только в период выполнения программы, вы не можете видеть их в тексте программы, хотя вы, конечно же, видите их имена, такие как *Paris* и *Route1,*обозначающие объекты, которые появятся во время исполнения программы.

Как следствие, термин «объект времени выполнения», появляющийся в определении класса, является избыточным, поскольку объекты по определению могут существовать только в период выполнения («run time»). С этого момента мы будем просто говорить «объект», не подчеркивая его динамическую сущность.

Поиск подходящих классов является центральной задачей проектирования ПО. Его цель: построить основную структуру программы – ее *архитектуру*. В противоположность этому, описание деталей относится к этапу, называемому *реализацией* (*implementation*).

4.3. Использование класса

Теперь мы постараемся понять, на что похож класс и как можно его использовать для построения новых классов – клиентских классов – в нашей собственной программе.

Уже знакомые нам классы были написаны для моделирования свойств и операций, связанных с транспортной системой города, подобной Парижскому метро. Мы используем Париж изза его известности – это город, являющийся мировым центром туризма. Но, конечно, это только пример, ПО ничего не знает об особенностях этого города. Вся информация о городе и его транспортной сети читается из файла, который можно заменять в соответствии с вашим выбором (в ETH мы используем Цюрих и его трамвайную сеть – основу транспорта в этом городе).

Файл, хранящий информацию, создается в формате XML – принятом ныне стандарте описания структурированных данных.

Ниже для ссылок приводится план Парижского метро (упрощенная версия, где опущены некоторые станции).

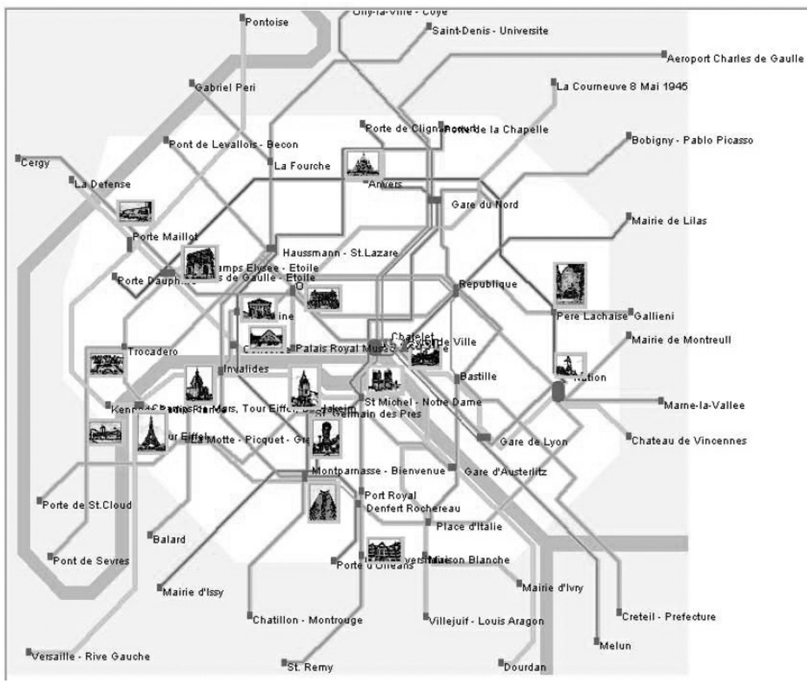


Рис. 4.2. План Парижского метро

Как определить, что должен делать хороший класс

Предположим, нас попросили спроектировать ПО, моделирующее метро, как оно видится нынешними и будущими пассажирами. Как и в любом проекте ПО, ключевым является вопрос: какие классы следует создать? Чтобы найти хорошие классы, отвечающие на наш вопрос, обратимся к поиску концепций проблемной области, которые:

- описывают множество объектов (их будущие экземпляры);
- могут быть четко объяснены;
- могут быть охарактеризованы в терминах четко определенных методов, включающих как запросы, так и команды, применимые к соответствующим объектам.

Документ с минимальными требованиями

Можем ли мы найти классы и их методы, моделирующие метро? Зачастую первый шаг проектирования состоит в том, чтобы описать простым и ясным языком проблемную область. Давайте попробуем.

Почувствуй Париж: Добро пожаловать в метро

> Метро – это сеть поездов, главным образом, идущая под землей, позволяющая людям путешествовать по городу быстро и с удобствами.
> Сеть состоит из линий, каждая линия включает множество станций, две из которых являются конечными станциями. Поезда на линии идут от одной конечной станции до другой, останавливаясь на каждой станции, расположенной вдоль линии, а затем возвращаются назад таким же способом.
> Некоторые станции принадлежат двум или более линиям; они называются пересадочными станциями, позволяя связывать одну линию с другой.
> Чтобы добраться до некоторого пункта назначения в городе, используя метро, необходимо идентифицировать станцию, ближайшую к той точке, где вы находитесь, и станцию, ближайшую к пункту назначения. После этого необходимо составить маршрут поездки между выбранными станциями. Маршрут включает несколько этапов, каждый из которых состоит из последовательно идущих станций одной линии. Последовательные этапы соединяются на пересадочных станциях. У метро есть важное свойство: всегда существует маршрут для любой пары станций метро (математики сказали бы, что метро задается *связным* графом).

Это описание довольно формальное, даже педантичное. Вряд ли его мог бы сформулировать посетитель, ранее не пользовавшийся метро. С другой стороны, оно менее точное и менее полное, чем можно было бы ожидать от «документа требований», используемого в индустрии при разработке проектов ПО (эта тема подробно обсуждается в главе, посвященной инженерии ПО). Данный текст требований достаточно хорош для наших текущих целей поиска нескольких классов.

Начальные идеи поиска классов

Как обычно, в документах требований некоторые детали не относятся к нашим непосредственным потребностям, например, то, что сеть главным образом располагается под землей. Само слово «сеть» оказывается не столь уже полезным. Но после некоторых размышлений можно выделить четыре концепции, претендующие на роль классов.

- *STATION*. Метро состоит из станций; люди едут от одной станции к другой, проезжая мимо некоторых станций. Это понятие кажется жизненно важным для нашего ПО.
- *LINE*. Метро состоит из линий, каждая из них соединяет несколько станций, проходящих в определенном порядке.
- *ROUTE*. Маршрут задает описание того, как добраться от одной станции метро к другой.
- *LEG*. Этап маршрута задает множество непосредственно следующих станций одной линии.

Между введенными понятиями существуют тесные отношения: линия составлена из станций, этап также состоит из станций и является частью линии, маршрут складывается из этапов разных линий.

В доступном для нас ПО Traffic есть классы, покрывающие эти понятия, так что мы сможем рассмотреть некоторые из их свойств. Более точно, Traffic – это **библиотека**: коллекция программных элементов, которые сами по себе не составляют программу, но обеспечивают важную функциональность, полезную для многих программ. Хотя библиотека Traffic доступна как часть материалов этой книги, в этой главе мы будем работать так, как если бы мы

обязаны были спроектировать соответствующие классы, начиная с базисных концепций: линия, станция, этап, маршрут. Конечно, вам не возбраняется изучать уже созданные классы Traffic. Если вы так и поступите, то учтите следующие соглашения.

Соглашения: Имена классов библиотеки Traffic

Для ясности и во избежание путаницы с другими библиотеками все классы в библиотеке Traffic имеют имена, начинающиеся префиксом *TRAFFIC_*, например, *TRAFFIC_STATION*, *TRAFFIC_LINE*, *TRAFFIC_ROUTE*, *TRAFFIC_LEG*. Для краткости обсуждение в этой книге игнорирует префикс, говоря о классах *STATION* или *LINE*. Префикс следует использовать при поиске библиотечных классов в EiffelStudio.

Это соглашение применимо к библиотечным классам. Имена классов в примере, такие как *TOURISM* и *PREVIEW* из предыдущей главы, не используют префикс.

Что характеризует линию метро

Давайте начнем с понимания интерфейса (API, программного интерфейса) класса *LINE*, представляющего линии метро. Вы можете использовать среду EiffelStudio, чтобы посмотреть интерфейс любого доступного класса.

Начнем со взгляда на упрощенную форму класса *LINE*, называемую *SIMPLE_LINE*. Что значит взглянуть на класс? Класс многолик, и на него можно смотреть с разных точек зре-

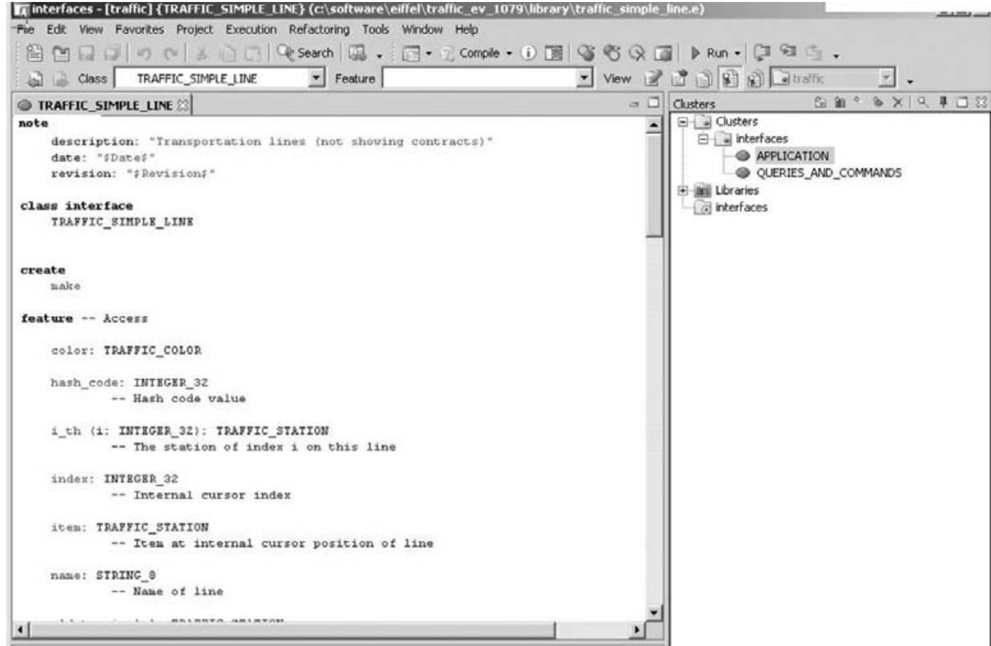


Рис. 4.3.

ния. Можно рассматривать текст класса, и EiffelStudio позволяет сделать это, но нам требуется прямо сейчас совсем иной взгляд. Помимо текста класса, EiffelStudio позволяет отображать другие *облики* (*views*) нашего класса, каждый из которых высвечивает различные свойства. Облик или вид класса, интересующий нас в данный момент, известен как **Контрактный Облик** (**contract view**). Причины такого наименования вскоре станут ясными.

Для его получения введите имя класса в соответствующее поле, затем щелкните кнопку «*contract view*», расположенную справа в верхнем ряду кнопок. Ее можно найти, подводя курсор к кнопке и следя за появляющейся подсказкой. Результат выглядит следующим образом:

Контрактный облик показывает методы – команды и запросы, – применимые к экземплярам класса *SIMPLE_LINE*, который представляет линию метро. Эти методы изучаются в последующих разделах.

> Сразу же после обсуждения методов вернитесь к предыдущей картинке (или снова отобразите на экране соответствующий контрактный облик), чтобы убедиться, что вы понимаете все детали контрактного облика класса.

Следя за обсуждением запросов и команд, следует помнить, что класс описывает множество возможных объектов, его экземпляров (каждый представляет линию метро). Методы объявляются в классе, каждый определяет операцию, применимую к любому такому объекту. Например, класс будет иметь запрос *south_end*, дающий одну из двух конечных станций (ту, которая южнее другой). Это означает, что мы можем применять этот запрос к любой линии метро. Если *Line8* обозначает экземпляр *SIMPLE_LINE*, то *Line8.south_end* обозначает ее конечную станцию. Когда речь не идет о конкретном экземпляре, мы позволяем себе говорить «*SIMPLE_LINE*», подразумевая под этим типичный экземпляр класса, задающего типичную линию.

Класс *SIMPLE_LINE* представляет слегка упрощенную версию класса LINE; обсуждение применимо к обоим классам. Рассмотрим вначале запросы, а потом перейдем к командам.

4.4. Запросы

Класс *SIMPLE_LINE* предлагает несколько запросов о характеристиках линии метро.

Сколько станций на линии?

Первое, что нам хотелось бы знать о линии, это сколько на ней станций, что обеспечивается запросом *count*. Спецификация этого запроса появляется в контрактном облике в виде:

```
count: INTEGER
        -- Число станций на этой линии
```

Вторая строка, как вы знаете, является комментарием, более точно – **заголовочным комментарием** (**header comment**), который должен сопровождать каждый компонент класса. Всегда полезно дать простое объяснение каждого компонента. Наряду с другими инструментами это позволяет избежать возможных недоразумений. Здесь, например, мы могли бы выбрать в качестве меры число сегментов, которое было бы на единицу меньше числа станций.

Комментарий проясняет наше соглашение: для класса *SIMPLE_LINE* запрос *count* задает число станций.

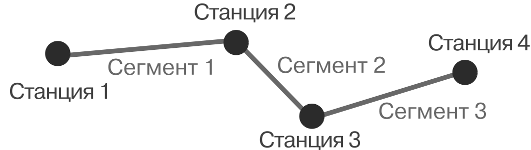


Рис. 4.4. 4 станции, 3 сегмента

Обратите внимание, в комментарии сказано: «на *этой* линии». Класс описывает общее понятие линии, но метод применяется к вполне конкретной линии, как в вызове *Line8.count*, и запрос будет возвращать в этом вызове число станций *Line8.* В конечном счете, класс всегда говорит о конкретной линии, даже если в классе эти конкретные линии нам не известны. Так что смысл «эта линия» в комментарии означает, что речь идет об объекте, к которому будет применяться метод *count*.

Объявление запроса начинается с *count: INTEGER*. Вначале вводится имя запроса, *count*, а затем тип возвращаемого результата, *INTEGER*. Запрос поставляет информацию об объекте, интерфейс класса должен определять ее тип.

Таким типом является *INTEGER*, обозначающий положительные и отрицательные целые значения, включая ноль. Другие типы, которые встретятся в этой главе, включают:

- *STRING*, для значений, представленных последовательностью символов, например, «*ABCDE*»;
- *BOOLEAN*, для «истинностных значений», которые могут быть только **True** или **False**;
- сами классы, такие как *STATION* или *LEG*.

Вскоре мы узнаем больше о типах, а пока достаточно и типа *INTEGER*.

Экспериментируем с запросами

Познакомившись с методами, можно испытать их.

Время программирования!

Длина линии

В первом упражнении на программирование в этой главе, детализированном ниже, требуется вычислить длину Линии 8.

Программная система, названная *interface* (в подкаталоге *04_interface*), является частью ПО Traffic. Откройте эту систему в EiffelStudio и рассмотрите класс *QUERIES_AND_COMMANDS:*

```
class QUERIES_AND_COMMANDS inherit
   TOURISM
feature
      tryout
            -- Испытайте запросы и команды, применимые к линии.
         do
            [          ]  <-- Часть, которую вы будете заполнять
         end
end
```

Рис. 4.5.

Этот класс является площадкой для проверки концепций этой главы. Заполните подсвеченную часть вызовами различных методов. Запустив полученную систему на выполнение, можно видеть возникающий в каждом случае эффект.

Нужные линии метро определены в контексте класса *TOURISM*. Введите в часть для заполнения следующий оператор:

```
Console.show (Line8.count)
```

В результате будет вызван только что описанный метод *count* для *Line8*, полученное значение будет передано команде *show* объекта *Console*, позволяющей отобразить результат в окне консоли. Так вы узнаете, сколько станций есть на Линии 8.

> Как и в главе по объектам, остается некоторая «магия», так как объекты Console и Line8 определены в данном нам классе TOURISM. Появится в этой главе и еще капелька магии, но нам необходимо сосредоточиться на новых концепциях. Довольно скоро магия исчезнет, и мы будем способны определять все, что нам необходимо.

В предыдущей главе мы говорили, что вызов *Line8*• *count*, означающий результат применения запроса к объекту, является выражением. Каждое выражение имеет тип. В данном случае тип выражения есть *INTEGER*, поскольку это тип результата, возвращаемого запросом *count*.

Станции на линии

Наши следующие запросы позволят нам узнать, какие же именно станции находятся на линии. Вспомните объяснение в нашем небольшом документе требований.

> Каждая линия содержит множество станций, две из которых являются конечными станциями

Хотя эта спецификация не совершенна и требует дополнения нашими интуитивными знаниями транспортной сети, из нее все же следует, что линия метро содержит последовательность станций: конечную станцию, станцию, еще одну станцию и так далее до следующей конечной станции. Простой способ представить это дает следующий запрос из нашего класса *SIMPLE_LINE*:

```
i_th (i: INTEGER): STATION
          -- Станция с индексом i на этой линии.
```

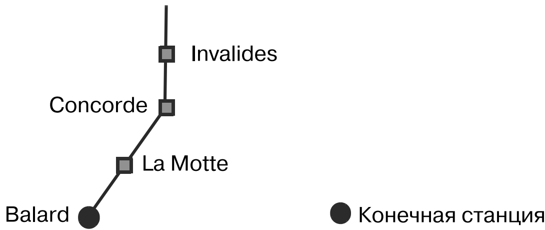


Рис. 4.6. Начало линии 8 (только основные станции, некоторые имена укорочены)

Имя *i_th* выбрано в соответствии с общим способом ссылки на элемент некоторой серии по его номеру (в английском – the 25-th element). Следует учесть, что при построении идентификаторов нельзя использовать тире, но можно использовать знак подчеркивания.

Запрос *i_th*, подобно *show* для *Console*, принимает аргумент – целое число, задающее индекс требуемой нами остановки, который имеет значение 1 для первой конечной станции, а затем увеличивается на единицу для каждой следующей станции. Снова возьмем в качестве примера Линию 8, представленную на рисунке 4.6.

Выражение

```
Line8.i_th (1)
```

представляет станцию «**Balard**»: *Line8.i_th* (2) – это станция «**La Motte**» и так далее. Будем использовать два соглашения.

Соглашения: нумерация станций на линии

- Линия метро всегда содержит, по крайней мере, одну станцию, даже если она пуста и имеет ноль сегментов.
- Нумерация станций на линии всегда начинается с южного конца. Запрос *south_end* будет обозначать эту станцию, *north_end* – другую конечную станцию. Для пустой линии или кольцевой оба запроса дают одну и ту же станцию. В тех редких случаях, когда конечные станции различны, но находятся на одной широте, *south_end* означает западную станцию.

Ссылка на пустые линии звучит странно, если иметь в виду существующие линии метро. Но мы моделируем абстрактное понятие «Линии», для которого пустая линия возможна. Позже в этой главе мы будем строить (виртуально) линию метро, начиная с пустой линии и добавляя в нее поочередно новые станции.

Класс *SIMPLE_LINE* имеет два запроса, обозначающие конечные станции на линии:

```
south_end: STATION
        -- Конечная станция на южной стороне.
north_end: STATION
        -- Конечная станция на северной стороне.
```

Время для теста

Другой конец

Line8.i_th (1) является выражением типа *STATION*, обозначающим станцию на южном конце Линии 8. Не используя число станций этой линии или имена станций (или ответ на этот тест, который появится ниже), напишите еще одно выражение, которое в том же стиле задает объект, представляющий станцию на другом конце линии. Подсказка: используйте другой уже введенный запрос.

Свойства начала и конца линий

Чтобы более точно выразить наше решение о начале нумерации с южного конца, заметим, что любая линия *l* удовлетворяет следующим свойствам:

```
l.south_end = l.i_th (1)
l.north_end = l.i_th (l.count)
```

Не стоит продолжать чтение, если вы в совершенстве не понимаете эти две программные строчки. Каждая устанавливает свойство линии *l* – эквивалентность, подобную эквивалентности в математике, такой как $cos^2(x) + sin^2(x) = 1$ для любого числа *x*.

- Первая эквивалентность говорит, что запрос *south_end* всегда возвращает тот же результат, что и запрос *i_th*, примененный к той же лини метро с аргументом 1. Другими словами, здесь устанавливается наше соглашение о нумерации станций, начиная с южного конца.
- Второе равенство устанавливает соответствующую эквивалентность для другого конца. Так как *l.count* обозначает число станций на линии, выражение *l.i_th* (*l.count*) обозначает последнюю станцию.

Соглашение, что линия всегда имеет станцию, даже если она пуста, также играет роль, поскольку в противном случае выражение *l.i_th* (1) не всегда имело бы смысл.

Все сказанное дает ответ на наш маленький тест: для получения ответа можно использовать выражение *Line8.i_th* (*Line8.count*) или более короткое – *l.north_end*.

4.5. Команды

До сих пор мы получали доступ к свойствам существующих линий, используя для этого запросы. Пришло время заняться другой категорией методов: командами, позволяющими изменять объекты.

Построение линии метро

Что можно делать с линией метро? Наиболее очевидный ответ – добавить новую станцию, например, к одному из ее концов.

Если вы думаете: «Это чепуха! Программа не может создавать станции метро, и линии метро уже существуют», – то вам, вероятно, полезно перечитать тот раздел, где объясняется, что наши объекты являются искусственными созданиями – артефактами – ПО, они не являются реальными вещами.

Давайте перестроим Линию 8. О классе *TOURISM* можно предположить следующее: предопределенные методы, такие как *Station_Balard*, *Station_La_Motte* и другие, доступны для каждой станции. Имя метода для станции «xxx» строится как *Station_Xxx*; Если имя содержит несколько слов, как для станции «La Motte», то слова разделяются подчеркиванием – *Station_La_Motte*.

Поскольку *Line8* предопределена в классе *TOURISM*, первое, что нам предстоит, – удалить из нее станции, сделав ее пустой. В контрактном облике класса *SIMPLE_LINE* вы могли заметить присутствие подходящей для этих целей команды:

```
remove_all_segments
        -- Удалить все станции, за исключением южного конца.
```

Наша программа будет использовать этот метод в вызове:

```
Line8.remove_all_segments
```

Напомним, что по соглашению наши линии всегда имеют по крайней мере одну станцию. Когда она только одна, как в результате вызова *remove_all_segments* (который, как ука-

зано в заголовочном комментарии, оставляет южную станцию), она является результатом обоих запросов *south_end* и *north_end*.

Теперь мы готовы добавлять станции. Снова вы можете видеть релевантные команды в контрактном облике класса:

```
extend (s: STATION)
        -- Добавить s в конец этой линии.
```

Это значит, что если *li* означает линию, то в ее конец можно добавить станцию *st*, благодаря вызову

```
li.extend (st)
```

Теперь можно заполнять текст примера этой главы:

```
class QUERIES_AND_COMMANDS inherit
   TOURISM
feature
      tryout
         -- Воссоздать частичную версию Линии 8.
         do
            Line8.remove_all_segments
                  -- Нет необходимости в добавлении Station_Balard, так как
                  -- удаляются все сегменты, оставляя южный конец.
            Line8.extend (Station_La_Motte)
            Line8.extend (Station_Concorde)
            Line8.extend (Station_Invalides)
                  -- Мы прекратим добавлять станции, чтобы отобразить
                  -- полученные результаты:
            Console.show (Line8.count)
            Console.show (Line8.north_end.name)
         end
end
```

Время теста

Имя последней станции

Как вы уже догадались, глядя на последний оператор, класс *STATION,* представляющий тип запроса *north_end*, имеет запрос *name*, который возвращает имя станции. Какое же имя отобразится в окне консоли при выполнении программы?

Чтобы убедиться в корректности ваших рассуждений, выполните этот пример.

4.6. Контракты

Одна из причин использования упрощенного класса вместо его финальной версии состоит в том, что мы опускаем одно фундаментальное свойство, которое никак нельзя игнориро-

вать при создании серьезного ПО. Дело в том, что не все методы принимают любые аргументы и не все применимы к любым экземплярам. Интерфейсы необходимы для уточнения того, что является допустимым.

Предусловия

Интерфейс для запроса *i_th* в классе *SIMPLE_LINE*, как показано ранее

```
i_th (i: INTEGER): STATION
            -- i-я станция на этой линии.
```

не упоминает, что только некоторые значения *i* имеют смысл: значение должно быть между 1 и числом станций на линии, *count*. Если Линия 8 имеет 20 станций, то было бы ошибкой использовать запрос *Line8*•*i_th* (300), или *Line8*•*i_th* (0), или *Line8*•*i_th* (–1).

> Если желаете, можете испытать такие выходящие за границы значения. Отредактируйте класс, выполните программу и посмотрите, что получится.

Программисту, который пытается понять, что делает некоторый класс, потенциальному «клиенту» класса такая информация необходима. И именно это дают интерфейсы – они говорят клиентам программистам, что данный класс может сделать для них.

Мы можем, конечно, добавить информацию в заголовочный комментарий, как в следующем фрагменте.

Предупреждение: нерекомендуемый стиль, смотри далее.

```
i_th (i: INTEGER): STATION
          –i-я станция на этой линии
          –(Предупреждение: используйте запрос только при i между 1 и count,
          – включительно)
```

Подобный комментарий лучше, чем ничего, но он не достаточно хорош. Такие свойства настолько общеупотребительны и настолько критичны для корректного использования классов и методов, что должны рассматриваться как неотъемлемая часть программы, того же уровня, что и операторы и выражения. Эти свойства называются **контрактами**. Для запроса введем наш первый элемент контракта – **предусловие**. Предусловие – это свойство, выполнение которого метод требует от всех своих клиентов, желающих вызывать метод; в данном случае – требование, чтобы аргумент находился в заданных границах.

Интерфейс метода показывает контракт, используя ключевое слово **require**. Поэтому контрактный облик класса *LINE* фактически описывает запрос *i_th* следующим образом:

```
i_th (i: INTEGER): STATION
            -- i-я станция на этой линии
        require
            not_too_small: i >= 1
            not_too_big: i <= count
```

Предложение предусловия состоит из двух раздельных элементов, называемых **утверждениями (assertions)**. Каждое выражает свойство: $i \geqslant 1$ в первом утверждении и $i \leqslant count$ – во

втором. Заметьте, из-за ограничений клавиатуры мы не можем для неравенств использовать обозначения, принятые в математике, и используем для неравенств два символа. Имена утверждений not_too_small и not_too_big называются **тегами утверждений** (**assertion tags**). Они служат для прояснения цели утверждений, но фактический смысл (семантика) следует из выражений: *i* >= 1 и *i* <= *count*. Мы можем опускать теги утверждений и двоеточие, как в следующем фрагменте:

```
require
    i >= 1
    i <= count
```

Смысл предусловий не меняется, но теги придают ясность, так что их необходимо включать, следуя правилам хорошего стиля.

Выражения, подобные *i* >= 1 и *i* <= *count*, обозначают условия, которые в любой момент выполнения программы могут быть либо истинными, либо ложными. Предыдущие примеры включали эквивалентность *l.south_end* = *l.i_th* (*1*), заданную для линии *l*. Выражения, которые имеют истинностные значения, записываемые в Eiffel как **True** и **False**, называются булевскими (**boolean**):

Определение: булевское значение

Булевское значение может быть либо **True**, либо **False**.

Соответствующий тип называется *BOOLEAN*. Это один из типов, имеющихся в нашем распоряжении наряду с *INTEGER*, *STRING* и именами определенных вами классов. Большинство типов, в отличие от булевского типа, имеют много значений, например, представление целого типа в компьютере поддерживает миллиарды возможных значений, но у типа *BOOLEAN* значений только два. Этот тип используется для представления условий, подобно тому, как это делается в обычном, не программистском мире («достаточно ли снега» – условие, позволяющее решить, стоит ли кататься на лыжах). Наши булевские выражения в мире ПО должны быть определены совершенно точно, как и в мире математики: выражение *i* >= 1 недвусмысленно имеет значение true или false, если известно значение *i*, в то время как условие «достаточно ли снега» субъективно и зависит от решения субъекта, собирающегося кататься на лыжах.

Булевские значения и булевские выражения лежат в основе *логики* – искусства точных рассуждений. Следующая глава будет посвящена этой теме.

Предусловия и другие формы контрактов будут использовать булевские выражения для задания условий, которые клиенты и поставщики должны выполнять. В предусловии запроса *i_th*, появляющемся в интерфейсе

```
require
    not_too_small: i >= 1
    not_too_big: i <= count
```

содержится важная информация для клиента.

Клиент, не удовлетворяющий свойству, например, вызывающий

```
Line8.i_th (1000)
```

создает неисправное ПО, пораженное «багом» («жучком»), где баг в компьютерной терминологии – аналог ошибки.

Мы можем выразить это наблюдение как некоторый общий принцип.

Почувствуй методологию:

Принцип Предусловия
Клиент, вызывающий метод, должен убедиться, что предусловие выполняется непосредственно перед вызовом.

Всякий раз, когда вы рассматриваете использование метода, необходимо ознакомиться с его спецификацией в контрактном облике соответствующего класса, включая предусловие, если оно задано, как в примере запроса *i_th*. На вас, как на программисте-клиенте ПО, лежит ответственность, что любой ваш вызов метода удовлетворяет предусловию метода.

Некоторые методы всегда применимы; они не имеют предложения **require**. По соглашению это эквивалентно присутствию предусловия в форме:

```
require
    always_OK: True
```

которое говорит, что предусловие всегда выполняется.

Контракты для отладки

Один из способов, благодаря которому предусловия и другие контракты помогают в разработке ПО, состоит в существовании инструментария, способного проверить контракты во время выполнения программы. Так что, если один из контрактов не работает, что свидетельствует о наличии жучка, то вы получите ясное сообщение, уведомляющее о случившемся. Сообщение укажет тег (такой как not_too_small) с нарушенным утверждением, и вы будете знать, что пошло не так, как требуется.

В программе, готовой для поставки – передачи ее клиентам, будучи уверенным, что к этому моменту все ошибки исправлены, можно отключить опцию проверки контрактов во время выполнения.

Раздел приложения EiffelStudio скажет вам, как настраивать параметры, определяющие, какие из контрактов будут проверяться во время выполнения.

Контракты для документирования интерфейса

Лучший способ обеспечения корректности ПО состоит в том, чтобы вообще не создавать жучков (а не в том, чтобы делать ошибки, а потом их исправлять). В этом вам поможет систематическое использование контрактов. В частности, механизм документирования ПО, задаваемый в интерфейсе, должен всегда перечислять полный список *предусловий*, определяющий, при каких условиях законно использовать тот или иной метод ПО.

Этот стиль, иллюстрируемый интерфейсом запроса *i_th*, станет стандартной формой описания интерфейса в оставшейся части этой книги. Это также тот вид представления интерфейса класса, который вы получаете, работая в EiffelStudio, – он, соответственно, назван *контрактным обликом* класса.

Постусловия

В описании интерфейса метода, предоставляемого потенциальным клиентам, предусловия задают только одну сторону отношений клиента и используемого ПО, а именно, что метод ожидает от клиента перед вызовом. Для клиентов предусловие является обязательством. Как и в любом хорошем отношении, помимо обязанностей есть и права, получаемые *преимущества* в результате вызова. Интерфейс метода позволяет выразить их в виде **постусловий**.

В отличие от предусловий, не всегда удается выражать постусловия полностью в виде релевантных свойств, но зачастую можно сказать много полезного. Вот пример интерфейса для метода *remove_all_segments* в классе *LINE*:

```
remove_all_segments
        -- Удалить все станции, кроме южного конца.
    ensure
        only_one_left: count = 1
        both_ends_same: south_end = north_end
```

Здесь нет предусловия, поскольку метод *remove_all_segments* всегда применим к маршруту. Ключевое слово **ensure** вводит постусловие. В нем метод гарантирует по окончании своей работы своим клиентам две вещи:

- число станций, *count*, будет равно 1;
- две конечные станции, *south_end* и *north_end*, теперь будут одной и той же станцией. Напомним, что это соответствует принятому соглашению: пустая линия не имеет сегментов; у нее только одна станция, которая является и южной, и северной.

Аналогичная ситуация (предусловие опущено) и для интерфейса команды *extend*, добавляющей станцию в конец линии:

```
extend (s: STATION)
        -- Добавить s в конец линии.
      ensure
        new_station_added: i_th (count) = s
        added_at_north: north_end = s
        one_more: count = old count + 1
```

Первое предложение постусловия использует запрос *i_th*. В нем утверждается, что если мы спросим после вызова команды *extend*, какая станция в позиции *count* (это эквивалентно вопросу, какая станция является последней), то ответ будет – *s*, станция, только что добавленная на линию. Так мы точно выражаем наше намерение относительно цели команды *extend*. Если текст команды написан без ошибок, то это свойство всегда будет существовать после окончания выполнения метода.

Второе предложение говорит, что после выполнения команды конечной станцией *north_end* также будет *s*. Из инварианта, с которым мы познакомимся в следующем разделе, следует, что *north_end* должно быть эквивалентно *i_th* (*count*), так что наше предложение фактически избыточно, но это не должно нас тревожить.

Третье предложение говорит нам, что метод увеличивает на единицу число станций на линии. Оно использует ключевое слово **old**, с которым мы еще не встречались. Постусловие устанавливает свойство, существующее при завершении вызова метода. Часто необходимо

связать значение выражения, полученного в результате выполнения, со значением *на входе* в процедуру. Отсюда следует полезность «Old» (старого выражения) в форме:

```
old some_expression
```

Это означает: «Значение *some_expression*, вычисленное в момент начала выполнения программы». Этим и определяется смысл предложения постусловия

```
count = old count + 1
```

Old-выражения и ключевое слово **old** могут появляться только в постусловиях.

Когда вы пишете метод класса, то можно предполагать, что предусловие выполняется перед началом выполнения, ответственность возлагается на клиента, но ваша обязанность – обеспечить выполнение постусловия в конце выполнения метода.

Почувствуй методологию:

Принцип Постусловия

Метод должен гарантировать, что если предусловие выполняется в начале выполнения, то постусловие будет выполняться в момент завершения работы метода.

Инварианты класса

Предусловия и постусловия являются логическими свойствами вашего ПО, каждое из них связано с определенным *методом*, скажем, *i_th*, *remove_all_segments* и *extend*, появлявшимся в вышеприведенных примерах.

Мы также используем логические свойства для характеристики класса в целом на уровне более высоком, чем уровень отдельных методов. Такое свойство известно как **инвариант класса**, оно выражает отношения между различными запросами класса.

Мы уже встречались с такими свойствами в примерах с линиями метро:

- соглашение о том, что линия метро содержит по меньшей мере одну станцию;
- наблюдение о свойствах линии *l*: *l*•*south_end* = *l*•*i_th* (1) и *l*•*north_end* = *l*•*i_th* (*l*•*count*).

Эти свойства верны **для всех** линий, а не только для конкретной линии *l*. Другими словами, они характеризуют класс в целом. Поэтому такие свойства являются инвариантами класса. Они появляются в конце текста класса в виде специальных предложений:

```
invariant
    at_least_one_station: count >= 1
    south_is_first: south_end = i_th (1)
    north_is_last: north_end = i_th (count)
    identical_ends_if_empty: (count = 1) implies (south_end = north_end)
```

Последнее выражение использует изучаемую в следующей главе логическую операцию **implies** (следование, импликация), из истинности импликации *a* **implies** *b* следует, что *b* имеет значение **True** всегда, когда *a* имеет значение **True**.

Этот пример типичен для роли инвариантов класса: выражает согласованные требования к запросам класса. Здесь эти требования отражают некоторую избыточность, существующую между запросами *south_end* и *north_end* класса *LINE*, которые поставляют информацию, которая также доступна через запрос *i_th*, применимый с аргументами 1 и *count*.

Другим примером мог бы служить класс *CAR_TRIP,* обеспечивающий запросы: *initial_odometer_reading*, *trip_time*, *average_speed* и *final_odometer_reading*. Их имена позволяют судить об их роли – « odometer reading » – это общее число пройденных километров во время путешествия. И снова здесь присутствует избыточность, отражаемая в инвариантах класса:

```
invariant
    consistent: final_odometer_reading = initial_odometer_reading +
                          trip_time _ average_speed
```

В принципе, нет ничего ошибочного в наличии избыточности запросов при проектировании класса. Все такие запросы могут быть полезными для клиентов класса, даже если они содержат одну и ту же внутреннюю информацию о соответствующих объектах. Но без инвариантов избыточность могла бы стать причиной недоразумений и даже ошибок. Инварианты четко и ясно отражают то, как разные запросы могут быть связаны друг с другом.

Ранее мы видели, что предусловия должны выполняться в точке начала вызова метода, а постусловия – в конце его работы. Инварианты, применимые ко всем методам класса, а не к отдельным представителям, должны существовать в обеих точках.

Почувствуй методологию: Принцип Инварианта класса

> Инвариант класса должен выполняться сразу же после создания объекта и существовать перед и после выполнения каждого из методов класса, доступных клиенту.

Контракты: определение

Мы уже видели различные виды контрактов – предусловия, постусловия, инварианты класса, из которых следует общее определение контракта.

Определение: Контракт

> Контракт – это спецификация свойств программного элемента, предназначенная для потенциальных клиентов.

Мы будем использовать контракты повсюду в ПО, с целью сделать ясным каждый его элемент – класс или метод. Как отмечалось, они служат для документирования ПО, особенно библиотек компонентов, предназначенных (подобно Traffic) для повторного использования многими приложениями. Контракты помогают в отладке, помогают избежать появления ошибок и, прежде всего, создавать корректное ПО.

4.7. Ключевые концепции, изученные в этой главе

- Программный элемент представляет один или несколько интерфейсов остальному миру.

- Классы существуют только в программных текстах. Объекты существуют только во время выполнения ПО.
- Класс описывает категорию возможных объектов.
- Каждый запрос возвращает результат, тип которого указан в объявлении запроса.
- Мы можем специфицировать интерфейс класса через «контрактный облик», который перечисляет все компоненты класса – команды и запросы – и для каждого из них свойства, важные для клиентов (других классов, использующих класс поставщика).
- Метод может иметь предусловия, которые специфицируют начальные свойства, позволяющие «законно» вызывать метод, и постусловия, которые задают финальные свойства, гарантируемые при завершении метода.
- Класс может иметь инварианты, которые специфицируют согласованные условия, связывающие значения различных запросов класса.
- Предусловия, постусловия и инварианты являются примерами контрактов.
- Наряду с другими применениями контракты помогают при проектировании ПО, его документировании и отладке.

Новый словарь

API	Абстрактный программный интерфейс	**Assertion**	Утверждение
		Assertion tag	Тег утверждения
Boolean	Булевский	**Bug**	Баг (жучок, ошибка)
Class invariant	Инвариант класса	**Client**	Клиент
Client Programmer	Клиент-программист	**Contract**	Контракт
Generating class	Генерирующий класс	**GUI**	Графический интерфейс пользователя
Implementation	Реализация	**Instance**	Экземпляр
Interface	Интерфейс	**Library**	Библиотека
Postcondition	Постусловие	**Precondition**	Предусловие
Program interface	Программный интерфейс	**Software design**	Проектирование ПО
Supplier	Поставщик	**Type**	Тип
User interface	Интерфейс пользователя		

4-У. Упражнения

4-У.1. Словарь

Дайте точные определения каждому из списка терминов, приведенных в словаре.

4-У.2. Карта концепций

Добавьте новые термины в карту концепций, спроектированную в предыдущих главах.

4-У.3. Нарушение контрактов

1. Напишите простую программу (начав с примера этой главы), в которой использовался бы запрос *i_th* класса *LINE*. Выполните его, используя известный *LINE*-объект, например *Line8*.

2. Измените аргумент, передаваемый запросу *i_th*, так, чтобы аргумент выходил за предписанные границы (меньше 1 или больше числа станций на линии). Выполните программу снова. Что произошло? Объясните сообщение, которое вы получите.

4-У.4. Нарушение инварианта

Инвариант должен выполняться при создании объекта, затем перед и после выполнения методов, вызванных клиентом. Однако не требуется выполнение инварианта *во время выполнения* метода. Можете ли вы придумать пример метода, при выполнении которого было бы разумно нарушить инвариант, а затем в конце восстановить его корректность?

4-У.5. Постусловие против инварианта

Возможно, вы не уверены, стоит ли включать некоторое условие в постусловие метода или включить его в инвариант класса. Какие критерии могут помочь в решении этой дилеммы?

4-У.6. Когда писать контракты

Примеры контрактов этой главы были добавлены к программным элементам – методам и классам – после того, как первая версия этих элементов стала доступной. Можете ли вы придумать обстоятельства, при которых предпочтительнее писать контракты перед реализацией соответствующих программных элементов?

5

Немного логики

Программирование в значительной степени связано с доказуемостью, с выводимостью. Мы должны иметь возможность понимать совсем не простое, проходящее через множество ветвлений поведение программ во время их выполнения. Человек физически не в состоянии проследить за мириадами базисных операций, выполняемых компьютером.

В принципе, все может быть выведено из текста программы простыми рассуждениями. Если бы существовала наука выводимости, она оказала бы нам существенную помощь.

Можно радоваться: есть такая наука – это логика. Логика – это механизм, стоящий за способностью человека делать выводы. Когда нам говорят, что Сократ – человек и все люди смертны, то без раздумья мы заключаем, что Сократ смертен. Сделать этот вывод нам помогли законы логики. Пусть справедливо утверждение: «Если температура в городе поднимается выше 30 градусов, то возникает угроза загрязнений». Кто-то говорит, что поскольку сегодня температура достигла только 28 градусов, угрозы загрязнений нет; мы скажем про такого человека, что его логика «хромает».

Логика – основа математики. Математики доверяют доказательствам в пять строчек или доказательствам, растянутым на 60 страниц, только потому, что каждый шаг доказательства выполнен в соответствии с правилами логики.

Логика – основа разработки ПО. Уже в предыдущей главе мы познакомились с условиями в контрактах, связанных с нашими классами и методами, например, в предусловии: «*i* должно быть между 1 и *count*». Мы будем также использовать условия, выражая действия, выполняемые программой, например: «Если *i* положительно, то выполни этот оператор».

Мы уже видели при изучении контрактов, что такие условия появляются в наших программах в форме «булевских выражений». Булевское выражение может быть сложным, включающим, например, операции «**not**», «**and**», «**or**», «**implies**». Это соответствует выводам, характерным для естественного языка: « Если пройдет 20 минут после назначенного срока, ***и*** она ***не*** позвонит ***или не*** пришлет SMS, то ***следует***, что она не появится совсем». Интуитивно мы прекрасно понимаем, что означает это высказывание, и этого понимания вполне достаточно и для условий, применяемых в ПО.

Но до определенных пределов. Разработка ПО требует доказуемости, а точные выводы требуют законов логики. Поэтому, прежде чем вернуться к нашим дорогим объектам и классам, следует более тесно познакомиться с законами логики.

Логика – *математическая логика*, если быть более точным, – самостоятельная наука, таковой является и ее ветвь «Логика в информатике», которой посвящены многие учебники и отдельные курсы. Я надеюсь, что такой курс вами уже пройден или будет пройден. Эта глава вводит основные понятия логики, необходимые в программировании. Хотя логика пользуется заслуженной славой как наука о выводимости, нам она нужна для более ограничен-

ных целей – для понимания той части выводимости, которая базируется на *условиях*. Логика даст нам прочную основу для выражения и понимания условий, появляющихся в контрактах и в операторах программы.

Первая часть главы вводит булеву алгебру в форме пропозиционального исчисления, которое имеет дело с базисными высказываниями, включающими специфические переменные. Вторая часть расширяет обсуждение до *логики предикатов*, позволяющей выражать свойства произвольного множества значений.

5.1. Булевские операции

Условие в булевой (булевской) алгебре, так же как и в языках программирования, выражается в виде булевского выражения, построенного из булевских переменных и операций. Результатом вычисления булевского выражения является булевское значение.

Булевские значения, переменные, операции, выражения

Существуют ровно две **булевские константы (boolean constants)**, также называемые «булевскими или истинностными значениями», которые будем записывать в виде **True** и **False** для совместимости с нашим языком программирования, хотя логики часто пишут просто **T** и **F,** а электротехники предпочитают обозначать их как 1 и 0.

Булевская переменная задается идентификатором, обозначающим булевское значение. Типично мы используем булевскую переменную, чтобы выразить свойство, которое может иметь в качестве значения истину или ложь. Говоря о погоде, мы могли бы ввести переменную *rain_today*, чтобы задать свойство, говорящее нам, будет ли сегодня идти дождь.

Начав с булевских констант и переменных, мы можем затем использовать булевские операции для получения булевских выражений. Например, если *rain_today* и *cuckoo_sang_last_night* («дождь сегодня» и «кукушка куковала сегодня ночью») являются булевскими переменными, то булевскими выражениями в соответствии с изучаемыми ниже правилами будут:

- *rain_today* – булевская переменная сама по себе без всяких операций является булевским выражением (простейшая форма наряду с булевскими константами);
- **not** *rain_today* – используется булевская операция **not**;
- (**not** *cuckoo_sang_last_night*) **implies** *rain_today* – используются операции **not** и **implies**, а также скобки для выделения подвыражений.

Каждая булевская операция, такая как **not**, **or**, **and**, =, **implies,** задает правила вычисления значения результирующего выражения по заданным значениям ее операндов.

В языке программирования булевские операции, подобно булевским константам, задаются ключевыми словами. В математических текстах знаки операций обычно задаются отдельными символами, не все из которых присутствуют на клавиатуре компьютера. Приведем соответствие между ключевыми словами и общеупотребительными символами в математике:

Ключевое слово Eiffel	Общеупотребительные математические символы
not	¬ или ~
or	∨ или |
and	∧ или &
=	⇔ или =
implies	=>

В Eiffel булевские константы, переменные и выражения имеют тип *BOOLEAN*, определенный классом, подобно всем типам. Класс *BOOLEAN* является библиотечным классом, доступным для просмотра в EiffelStudio; там вы можете увидеть все булевские операции, обсуждаемые в этой главе.

Отрицание

Рассмотрим первую операцию – **not**. Для формирования булевского выражения с **not** укажите эту операцию с последующим булевским выражением. Это выражение может быть булевской переменной, как в **not** *your_variable*; или может быть сложным выражением (заключенным в скобки для устранения двусмысленности), как в следующих примерах, где *a* и *b* являются булевским переменными:

- **not** (*a* **or** *b*).
- **not** (**not** *a*)

Для произвольной булевской переменной *a* значение **not** *a* есть **False**, если значение *a* есть **True**, и **True**, если значение *a* – **False**, мы можем выразить свойства операции **not** следующей таблицей:

a	**not** *a*
True	**False**
False	**True**

Это так называемая **таблица истинности** (**truth table**), которая является стандартным способом задания булевских операций – в первых столбцах (для данной операции один столбец) задаются все возможные значения операндов операции, в последнем столбце задается соответствующее данному случаю значение выражения.

Операция **not** задает **отрицание** – замену булевского значения его противоположностью, где **True** и **False** противоположны друг другу.

Из таблицы истинности следуют важные свойства этой операции.

Теоремы: «Свойства отрицания»

Для любого булевского выражения *e* и любых значений входящих в него переменных:

- точно одно из выражений *e* или **not** *e* имеет значение **True**;
- точно одно из выражений *e* или **not** *e* имеет значение **False**;
- только одно из выражений *e* или **not** *e* имеет значение **True (принцип исключенного третьего)**;
- Либо *e,* либо **not** *e* имеет значение **True (принцип непротиворечивости)**

Доказательство: по определению булевского выражения, *e* может иметь только значение **True** или **False**. Таблица истинности показывает, что если *e* имеет значение **True**, то **not** *e* будет иметь значение **False**; все четыре свойства являются следствиями этого факта (и два последних утверждения следуют непосредственно из первого).

Дизъюнкция

Операция **or** использует два операнда в отличие от операции **not**. Если *a* и *b* являются булевскими выражениями, булевское выражение *a* **or** *b* имеет значение **True**, если и только если либо *a*, либо *b* имеют это значение. Соответственно, оно имеет значение **False**, если

и только если оба операнда имеют это значение. Это отражает следующая таблица истинности:

a	*b*	*a* **or** *b*
True	**True**	**True**
True	**False**	**True**
False	**True**	**True**
False	**False**	**False**

Первые два столбца перечисляют все четыре возможные комбинации значений *a* и *b*.

Слово «or» заимствовано из естественного языка в его **неисключительном** смысле, как в предложении «Тот, кто придумал эту инструкцию, глуп или слеп», что не исключает, что оба случая могут иметь место.

Обычный язык часто использует «или» в исключающем смысле, означающее, что только один результат может быть истинным, но не оба вместе: «Что будем заказывать – красное или белое?». Здесь речь идет о другой булевской операции – «исключающему или» – «**xor**» в Eiffel, чьи свойства следует изучить самостоятельно.

Неисключающая операция **or** называется **дизъюнкцией**. Это не очень удачное имя, поскольку позволяет думать об исключающей операции, но в нем есть свое преимущество, благодаря симметрии с «конъюнкцией» – именем следующей операции **and**.

Дизъюнкция имеет значение **False** только в одном из четырех возможных случаев, заданном последней строкой таблицы истинности.

Теорема: «Принцип дизъюнкции»

> Дизъюнкция **or** имеет значение **True**, за исключением случая, когда оба операнда имеют значение **False**.

Таблица истинности показывает, что операция **or** является *коммутативной*: для любых *a* и *b* значение *a* **or** *b* то же, что и *b* **or** *a*. Это также вытекает из принципа дизъюнкции.

Конъюнкция

Подобно **or**, операция **and** имеет два операнда. Если *a* и *b* являются булевскими выражениями, то булевское выражение *a* **and** *b* имеет значение **True,** если и только если *a* и *b* оба имеют это значение. Соответственно, оно имеет значение **False,** если и только если хотя бы один из операндов имеет это значение. Это отражено в таблице истинности.

a	*b*	*a* **and** *b*
True	**True**	**True**
True	**False**	**False**
False	**True**	**False**
False	**False**	**False**

Применение операции **and** к двум значениям известно как их **конъюнкция**, аналог конъюнкции двух событий: «*Только конъюнкция* (*сочетание*) *полной луны и низкой орбиты Сатурна принесет Стрельцам истинное любовное приключение*» (возможно, гороскопы не есть главное дело для логиков).

Изучение **and** и **or** показывает их близкое соответствие – **двойственность**. Многие свойства одной операции могут быть получены из свойства другой простым обменом **True** и **False**. Например, принцип дизъюнкции двойственно применим к конъюнкции.

Теорема: «Принцип конъюнкции»

> Операция **and** имеет значение **False**, исключая тот случай, когда оба операнда имеют значение **True**.

Подобно **or**, операция **and** *коммутативна*: для любого *a* и *b*, *a* **and** *b* имеет значение такое же, как *b* **and** *a*. Это свойство следует из таблицы истинности, а также является следствием принципа конъюнкции.

Сложные выражения

Вы можете использовать булевские операции – три уже введены, **not**, **or** и **and**, а две будут описаны ниже – для построения сложных булевских выражений. Для таких выражений также можно задать таблицу истинности. Вот пример такой таблицы для выражения *a* **and** (**not** *b*):

a	*b*	**not** *b*	*a* **and** (**not** *b*)
True	**True**	**False**	**False**
True	**False**	**True**	**True**
False	**True**	**False**	**False**
False	**False**	**True**	**False**

Для порождения этой таблицы достаточно заменить в истинностной таблице для **and** каждое значение *b* значением **not** *b,* полученным из истинностной таблицы для **not**; третий столбец добавлен, чтобы показать **not** *b*.

Истинностное присваивание

Булевская переменная может принимать значение **True** или **False**. Значение булевского выражения зависит от значений его переменных. Например, при построении истинностной таблицы для выражения *a* **and** (*b* **and** (**not** *c*)) мы видим, что это выражение имеет:

- значение **True** если *a* и *b* имеют значение **True**, а *c* имеет значение **False**;
- значение **False** во всех остальных случаях.

Следующее понятие помогает выразить такие свойства.

Определение: Истинностное присваивание

> **Истинностным присваиванием** (truth assignment) для множества переменных называется индивидуальный выбор значений **True** или **False** для каждой из переменных.

Так что мы можем сказать, что выражение *a* **and** (*b* **and** (**not** *c*)) имеет значение **True** в точности для одного истинностного присваивания (**True** для *a*, **True** для *b*, и **False** для *c*) и **False** для всех остальных истинностных присваиваний.

Каждая строка истинностной таблицы некоторого выражения соответствует, один в один, истинностному присваиванию его переменных.

Достаточно просто заметить, что для выражения, содержащего *n* переменных, существует ровно 2^n истинностных присваиваний и, следовательно, 2^n строк в таблице истинности. Например, таблица для **not** с одним операндом имеет $2^1 = 2$ строки, таблица для **or** с двумя операндами имеет $2^2 = 4$ строки. Число столбцов в такой таблице равно $n + 1$.

- Первые *n* столбцов каждой строки перечисляют значения каждой из переменных соответствующего истинностного присваивания.

- Последний столбец дает значение выражения, соответствующее этому истинностному присваиванию.

(Только для целей пояснения в таблицу последнего примера добавлен столбец для **not** *b*)

Если для некоторого истинностного присваивания выражение имеет значение **True**, то будем говорить, что истинностное присваивание **удовлетворяет** выражению.

Например, уже рассмотренное присваивание – **True** для *a*, **True** для *b*, **False** для *c* – удовлетворяет *a* **and** (*b* **and** (**not** *c*)); все остальные – нет.

Тавтологии

Нас часто будут интересовать выражения, имеющие значение **True** для всех истинностных присваиваний. Рассмотрим

```
a or (not a)
```

Истинность этого выражения следует из того, что истинно одно из двух утверждений:

- *a* имеет значение **True**;
- **not** *a* имеет значение **True**.

Это неформальная интерпретация. Формально следует построить таблицу истинности.

a	**not** *a*	*a* **or (not** *a***)**
True	**False**	**True**
False	**True**	**True**

Строго говоря, второй столбец не является частью таблицы истинности; он дает значение для **not** *a*, приходящее из таблицы истинности для **not**. Комбинируя эти значения с таблицей истинности для **or**, получаем значения третьего столбца.

Из этого столбца видим, что любое истинностное присваивание (в данном случае переменная только одна) переменной *a* удовлетворяет выражению. Выражения с таким свойством имеют собственное имя.

Определение: Тавтология

Тавтологией называется булевское выражение, принимающее значение True для всех возможных истинностных присваиваний переменным этого выражения.

То, что выражение *a* **or** (**not** *a*) является тавтологией, ранее было установлено как принцип исключенного третьего.

Другими простыми тавтологиями, которые вам следует проверить, построив их таблицу истинности, являются:

- **not** (*a* **and** (**not** *a*)), выражающее принцип непротиворечивости;
- (*a* **and** *b*) **or** ((**not** *a*) **or** (**not** *b*))

Двойственными к тавтологиям являются выражения, которые *никогда* не принимают значения **True**.

Определение: Противоречие

Противоречием называется булевское выражение, принимающее значение **False** для каждого истинностного присваивания его переменным.

Например, (снова проверьте таблицу истинности) *a* **and** (**not** *a*) является противоречием. В другой формулировке об этом же говорит принцип непротиворечивости.

Из этих определений и таблицы истинности для **not** следует, что выражение *a* является тавтологией, если и только если **not** *a* является противоречием. Справедливо и обратное утверждение.

Выражение, имеющее значение **True** хотя бы для одного истинностного присваивания, называется **выполнимым**. Очевидно:

- любая тавтология выполнима;
- никакое противоречие не является выполнимым.

Существуют, конечно же, выполнимые выражения, не являющиеся тавтологиями. Они имеют значение **True** по крайней мере для одного истинностного присваивания и значение **False** для другого хотя бы одного присваивания. Таковыми являются, например, выражения *a* **and** *b* и *a* **or** *b*.

> Из того, что «*a* не является тавтологией», вовсе не следует, что «**not** *a* является тавтологией». Выражение *a* не является тавтологией, но может быть выполнимым, а значит, существует такое истинностное присваивание, при котором *a* примет значение **True**, но для этого же присваивания выражение **not** *a* будет иметь значение **False** и, следовательно, не будет являться тавтологией.

Эквивалентность

Для доказательства или опровержения тавтологий, противоречий, выполнимости построение таблиц истинности с их 2^n строками – дело весьма утомительное. Нам нужен лучший путь. Пришло время для общих правил.

Операция *эквивалентности* поможет определить такие правила. Она использует символ равенства «=» и имеет таблицу истинности (и это будет последняя таблица истинности), устанавливающую, что $a = b$ имеет значение **True,** если и только если *a* и *b* оба имеют значение **True** либо оба имеют значение **False**.

a	*b*	*a* = *b*
True	**True**	**True**
True	**False**	**False**
False	**True**	**False**
False	**False**	**True**

Операция коммутативна ($a = b$ всегда имеет то же значение, что и $b = a$). Операция *рефлексивна*, и это означает, что $a = a$ является тавтологией для любого *a*.

Хотя логики чаще используют для эквивалентности символ _, символ равенства также является подходящим, поскольку равенство в обычном смысле подразумевает эквивалентность: выражение $a = b$ имеет значение **True**, если и только если *a* и *b* имеют одинаковые значения. Следующее свойство расширяет это наблюдение.

Теорема: «Подстановка»

> Для любых булевых выражений *u*, *v* и *e*, если *u* = *v* является тавтологией и *e’* – это выражение, полученное из *e* заменой каждого вхождения *u* на *v*, то *e* = *e’* – это тавтология.

Набросок доказательства: если *u* не содержится в *e*, то *e'*совпадает с *e*, и по свойству рефлексивности *e* = *e* является тавтологией. Пусть теперь *u* входит в *e.* Заметим, что для каждого истинностного присваивания значение выражения полностью определяется значениями входящих в него подвыражений. Выражения *e* и *e'* отличаются только подвыражениями *u* и *v*, которые для данного присваивания имеют совпадающие значения, ввиду тавтологии *u* = *v*. Поскольку все подвыражения совпадают, будут и совпадать сами выражения *e* и *e'*. Данное рассуждение справедливо для любого истинностного присваивания. Отсюда следует, что на всех присваиваниях выражения *e* и *e'* имеют совпадающие значения, а следовательно, *e* = *e'* является тавтологией.

Это правило является ключом к доказательству нетривиальных булевских свойств. При доказательстве мы будем применять таблицы истинности только для базисных выражений, а затем, используя эквивалентность, заменим выражения на более простые. Например, рассмотрим выражение:

```
(a and (not (not b))) = (a and b)                    – Цель
```

Для доказательства того, что это выражение является тавтологией, нет необходимости выписывать таблицу истинности. Докажем прежде, что для любого выражения *x* следующие общие свойства являются тавтологиями:

```
not (not x) = x                              T1
x = x                                            T2
```

Утверждение T2 следует из рефлексивности эквивалентности =. Доказать T1 просто. Затем можно применить T1 к выражению *b* и использовать теорему о подстановке, заменяя **not** (**not** *b*) на *b* в левой части выражения «Цель». После чего остается применить T2 к *a* **and** *b*, получив желаемый результат.

Мы будем использовать пару символов «/=» для выражения того, что два булевских значения не эквивалентны. По определению *a* /= *b* имеет то же значение, что и **not** (*a* = *b*).

Законы Де Моргана

Две тавтологии представляют особый интерес в использовании **and**, **or** и **not**.

Теоремы: «Законы Де Моргана»

Следующие два свойства являются тавтологиями:

- (**not** (*a* **or** *b*)) = ((**not** *a*) **and** (**not** *b*))
- (**not** (*a* **and** *b*)) = ((**not** *a*) **or** (**not** *b*))

Доказательство: либо напишите таблицы истинности, либо, что лучше, комбинируйте принципы исключенного третьего, непротиворечивости, дизъюнкции и конъюнкции.

Эти свойства четко проявляют двойственность, характерную для операций **and**-**or**. Отрицание одной из этих операций эквивалентно применению другой операции над отрицаниями операндов.

Неформальная интерпретация: «Если кто-то говорит, что неверно, что имеет место *a* or *b*, то это все равно, если бы он сказал, что ни *a,* ни *b* не выполняются».

Конечно, мы уже находимся на этапе, когда формальные нотации с их точностью и лаконичностью значительно превосходят предложения естественных языков.

Другой аспект тесной ассоциации между операциями **or** и **and** состоит в том, что каждая из них **дистрибутивна** по отношению к другой. Смысл этого отражается в следующих двух тавтологиях.

Теоремы: «Дистрибутивность булевских операций»

Следующие два свойства являются тавтологиями:

- (*a* **and** (*b* **or** *c*)) = ((*a* **and** *b*) **or** (*a* **and** *c*))
- (*a* **or** (*b* **and** *c*)) = ((*a* **or** *b*) **and** (*a* **or** *c*))

Сравните с дистрибутивностью умножения по отношению к сложению в математике: для любых чисел m, p, q выражение $m \times (p + q)$ эквивалентно $(m \times p) + (m \times q)$.

Доказать дистрибутивность достаточно просто, например, через таблицы истинности. Она помогает упростить сложные булевские выражения.

Упрощение нотации

Во избежание накопления скобок для операций задаются правила приоритета, что обеспечивает стандартное понимание булевских выражений, когда опущены некоторые скобки. Идея здесь та же, что и в арифметике и языках программирования. Благодаря приоритетам мы понимаем, что в арифметическом выражении $m + p \times q$ первой будет выполняться операция умножения, имеющая более **высокий приоритет**. Иногда говорят, что операция умножения **теснее связывает** операнды, чем операция сложения.

Приоритеты булевских операций отражены в синтаксисе Eiffel. Порядок, задающий приоритеты от высшего к низшему, следующий:

- **not** связывает наиболее тесно;
- затем идет эквивалентность =;
- потом идет **and**;
- потом **or**;
- затем уже **implies** (изучаемая ниже).

По этим правилам выражение без скобок *a* = *b* **or** *c* **and not** *d* = *e* также законно и означает то же, что и выражение с расставленными скобками

```
(a = b) or (c and ((not d) = e))
```

Желательно все же оставлять некоторые скобки, защищая читателя от возможного непонимания, которое может привести к ошибкам.

В рекомендуемом стиле не следует опускать скобки, разделяющие **or** и **and** выражения, поскольку правила приоритета, устанавливающие, что **and** связывает теснее, чем **or**, являются спорными. Лучше сохранять скобки для **not** подвыражения, используемого как операнд эквивалентности, чтобы не путать (**not** *a*) = *b* с **not** (*a* = *b*). Можно опускать скобки для эквивалентности в форме $x = y$, где x и y являются простыми переменными. С учетом этих правил последний пример можно записать проще:

```
a = b or (c and (not d) = e)
```

Еще одно свойство, упрощающее нотацию, связано с **ассоциативностью** некоторых операций. В арифметике мы просто пишем $m + p + q$, хотя это могло бы означать $m + (p + q)$ или $(m + p) + q$, поскольку выбор порядка сложения не имеет значения. При любом порядке результат будет одинаков, потому что операция сложения обладает свойством ассоциативности. Этим же свойством в арифметике обладает операция умножения, но им не обладает операция деления. Для умножения два выражения $m \times (p \times q)$ и $(m \times p) \times q$ дают одинаковый результат. В булевской логике обе операции **and** и **or** являются ассоциативными, что выражается следующими тавтологиями:

```
(a and (b and c)) = ((a and b) and c)
(a or (b or c) = ((a or b) or c)
```

Указание для доказательства: можно, конечно, построить таблицу истинности, но проще использовать предыдущие правила. Для доказательства первой тавтологии следует использовать принцип конъюнкции. Дважды применяя это принцип к левой части эквивалентности, устанавливаем, что левая часть истинна, если и только если все переменные, в нее входящие, имеют значение true. Аналогичный результат справедлив и для правой части. Следовательно, обе части эквивалентности дают одинаковые результаты на всех истинностных присваиваниях и выражение является тавтологией. Вторая тавтология доказывается аналогично с использованием принципа дизъюнкции.

Это позволяет нам писать выражения в форме *a* **and** *b* **and** *c* или *a* **or** *b* **or** *c* без риска появления недоразумений. Обобщая:

Почувствуй стиль

Правила расстановки скобок в булевских выражениях

При записи подвыражений булевского выражения опускайте скобки:

- вокруг «*a* = *b*», если *a* и *b* – это простые переменные;
- вокруг последовательно идущих термов, если они состоят из булевских переменных, разделенных одинаковыми ассоциативными операциями.

Для повышения ясности записи и во избежание возможных ошибок оставляйте другие скобки, не надеясь на преимущества, предоставляемые приоритетами операций.

5.2. Импликация

Еще одна базисная операция, принадлежащая основному репертуару: **импликация**. Хотя она схожа с другими операциями **not**, **or**, **and** и эквивалентностью близка к операции **or**, она требует особого внимания, поскольку некоторым людям кажется, что ее точные свойства противоречат здравому смыслу и понятию следования в естественном языке.

Определение

Простейший способ определения операции **implies** состоит в том, чтобы выразить ее в терминах уже определенных операций **or** и **not**.

Определение: Импликация

Значение *a* **implies** *b* для любых булевских значений *a* и *b* является значением (**not** *a*) **or** *b*

Это временное определение. Позже будет дано еще одно определение.

Приведенное определение позволяет построить таблицу истинности (которая сама по себе может служить определением).

a	*b*	*a* **implies** *b*
True	**True**	**True**
True	**False**	**False**
False	**True**	**True**
False	**False**	**True**

Нетрудно видеть, что это таблица для **or**, в которой в столбце для *a* значения **True** и **False** поменялись местами. Результат *a* **implies** *b* истинен для всех истинностных присваиваний, за исключением одного случая, когда *a* равно true, а *b* – false.

В импликации *a* **implies** *b* первый операнд *a* называется **посылкой** (**antecedent**), второй, *b*, – **следствием** или **заключением** (**consequent**).

Принципы, которые были установлены для конъюнкции и особенно для дизъюнкции, имеют прямой аналог и для импликации.

Теорема: «Принцип Импликации»

Импликация имеет значение **True** за одним исключением, когда посылка имеет значение **True**, а заключение – **False**.
Как следствие, она имеет значение **True**, когда выполняется одно из двух условий:
I1 посылка имеет значение **False**;
I2 заключение имеет значение **True**.

Связь с выводами

Имя операции «**implies**» (влечет) предполагает, что эту операцию можно использовать для вывода одних свойств из других. Это и в самом деле допустимо, что и устанавливается следующей теоремой.

Теорема: «Импликация и вывод»

I3 Если истинностное присваивание удовлетворяет как *a*, так и *a* **implies** *b*, то оно удовлетворяет и *b*.
I4 Если оба *a* и *a* **implies** *b* являются тавтологиями, *b* – также тавтология.

Доказательство: Для доказательства I3 рассмотрим истинностное присваивание *TA*, удовлетворяющее *a*. Если *TA* также удовлетворяет *a* **implies** *b*, то оно должно удовлетворять и *b*, так как в противном случае из второй строки истинностной таблицы **implies** следовало бы, что *a* **implies** *b* равно **False**. Для доказательства I4 заметьте, что если *a* и *a* **implies** *b* являются тавтологиями, то предыдущий вывод распространяется на любое истинностное присваивание *TA*.

Это свойство делает законным обычную практику вывода. Когда мы хотим доказать истинность свойства *b*, то вводим более «сильное» свойство *a* и независимо доказываем истинность двух утверждений:

- *a*;
- *a* **implies** *b*.

Отсюда следует истинность *b*.

Использованный термин «сильнее» полезен в практике обоснования контрактов в программах и заслуживает точного определения.

Определения: сильнее, слабее

Для двух неэквивалентных выражений *a* и *b* мы говорим, что:

- «*a* **сильнее** *b*», если и только если *a* **implies** *b* является тавтологией;
- «*a* **слабее** *b*», если и только если *b* **сильнее** *a*.

Определения предполагают, что *a* и *b* не являются эквивалентными, поскольку некорректно говорить, что *a* «сильнее» *b*, если они одинаковы. Для случаев возможного равенства будем говорить «сильнее или эквивалентно», «слабее или эквивалентно» (подобно отношениям над числами «больше», «больше или равно»).

Практическое ощущение импликации

Какова связь операции **implies** с обычным понятием следования, выражаемого в естественном языке таким словосочетанием, как «Если ..., то...»?

В повседневном использовании импликация зачастую задает причинную связь: «Если лето будет солнечным, то это будет удачей для виноградников Бургундии» – это предполагает, что одно событие является причиной другого. В логике **implies** не ассоциируется с причинностью. Импликация просто устанавливает, что когда одно свойство является истинным, таковым должно быть и другое свойство. Приведенный пример допускает такую интерпретацию, если устранить всякий намек на причинность.

Вот еще один типичный пример (в аэропорту Лос-Анджелеса при попытке зарегистрироваться на рейс до Санта-Барбары): «Если на вашем билете напечатано "Рейс 3035", то сегодня вы не летите». Вероятно, причина отмены рейса связана с неисправностью самолета, и это был последний рейс на сегодняшний день. Понятно, что нет причинной связи между тем, что напечатано на билете, и отменой рейса. Логическая операция **implies** предусматривает такие сценарии.

Что удивляет многих людей, так это свойство I1 принципа импликации, следующие из двух последних строчек таблицы истинности. Когда *a* ложно, то *a* **implies** *b* истинно, независимо от значения *b*. Но на самом деле это соответствует обычной идее импликации:

1) «Если я губернатор Калифорнии, то дважды два – пять»;
2) «Если дважды два – пять, то я губернатор Калифорнии»;
3) «Если дважды два – пять, то я не губернатор Калифорнии»;
4) «Если я губернатор Калифорнии, то дважды два – четыре»;
5) «Если я губернатор Калифорнии, то сегодня пойдет дождь»;
6) «Если сегодня пойдет дождь, то я не буду выбран губернатором Калифорнии».

Если я признаю, что я не губернатор и не собираюсь стать им, все импликации будут истинными, невзирая на сегодняшнюю погоду.

Из истинности импликации следует только то, что если посылка верна, то это же верно и для заключения. Единственной возможностью для импликации быть ложной является слу-

чай истинной посылки и ложного заключения. Случаи, когда посылка ложна или заключение истинно, не определяют импликацию полностью, оставляя возможность неопределенности второго операнда.

Начинающим иногда трудно принять, что *a* **implies** *b* может быть истинным, если *a* ложно. Во многом, я полагаю, трудности связаны со случаем, в котором *a* ложно, а *b* истинно, – таким, как случаи 1, 2, возможно, 5 и 6 из приведенного выше примера. Но ничего ошибочного в них нет. Непонимание может быть связано с общим искажением правил вывода, допускаемым некоторыми людьми. Из истинности импликации a ***implies*** b, и зная, что *a* не выполняется, они без раздумья полагают, что и *b* не выполняется! Вот типичные примеры:

1. «Все профессиональные политики – коррупционеры. Я не политик, так что я не коррупционер, и вы должны голосовать за меня». Первое высказывание что-то утверждает о политиках, но оно не содержит информации о тех, кто политиками не является.
2. «Всегда, когда я беру с собой зонтик, дождь не идет. Я оставлю свой зонтик дома, поскольку нам так нужен дождь». Шутка, конечно, но логика явно хромает.
3. «Все недавно построенные здания в этом районе имеют плохую термоизоляцию. Это старое здание, так что в нем более комфортно в жаркие дни».

Каждый из случаев включает свойство *a*, которое влечет другое свойство *b,* и ошибочный вывод, что отрицание *a* влечет отрицание *b*. Это недопустимые выводы. Все, что мы знаем: если *a* верно, то и *b* верно. Если же *a* невыполнимо, то импликация нам ничего интересного не добавляет. Переходя на язык логики: ошибочно утверждение, что выражение

```
(a implies b) = ((not a) implies (not b))
```

является тавтологией. Более слабое утверждение

```
(a implies b) implies ((not a) implies (not b))
```

также не является тавтологией. Задайте два истинностных присваивания, при которых оба утверждения принимают значения **True** и **False** и, следовательно, являются выполнимыми, но не являются тавтологиями.

Обращение импликации

Хотя два последних свойства не являются тавтологиями, однако имеется интересная тавтология того же общего стиля:

```
(a implies b) = ((not b)) implies (not a))    – РЕВЕРС
```

Доказательство: Вспомним определение импликации. Тогда левая сторона дает (**not** *a*) **or** *b*, а справа имеем (**not** (**not** *b*)) **or** (**not** *a*). Из предыдущих тавтологий известно, что (**not** (**not** *b*)) = *b*. Учитывая коммутативность **or**, получаем нужный результат.

> Другой способ. В таблице истинности для **implies** меняем столбцы для *a* и для *b* местами, одновременно выполняя инверсию значений. Приходим к исходной таблице.

По свойству РЕВЕРС если *b* верно всякий раз, когда *a* верно, то: если *b* не выполняется, то не будет выполняться и *a*.

Неформальное доказательство от противного: если бы *a* было истинным, то импликация говорит нам, что и *b* было бы истинным, а оно ложно. Пришли к противоречию, значит, посылка неверна и *a* ложно.

Используя это правило, мы можем заменить ранее хромавшие выводы логически верными рассуждениями.

1. Все профессиональные политики коррупционеры. Она не коррупционер, поэтому она не является профессиональным политиком.
2. Всегда, когда беру с собой зонтик, дождя не бывает. Хотя на сайте weather.com обещают столь нужный нам дождь, на всякий случай я оставлю зонтик дома.
3. Все недавно построенные здания в этом районе имеют плохую термоизоляцию. В этой квартире, несмотря на жару, прохладно. Дом должен быть старым.

5.3. Полустрогие булевские операции

Математическая логика лежит в основе программирования, так что некоторые ученые полагают, что программирование – это просто расширение логики. Все наиболее замечательные факты, установленные современной логикой, появились в первые десятилетия двадцатого века, еще до наступления компьютерной эры в современном понимании.

Почувствуй историю

Дорога к современной логике

Логика корнями уходит в прошлые века, в частности, к Аристотелю, который определил правила «Риторики», зафиксировав некоторые формы вывода. В 18-м веке Лейбниц установил, что вывод – это форма математики. В 19-м веке английский математик Джордж Буль определил исчисление истинностных значений (в его честь – тип «boolean»). В следующем столетии большим толчком в развитии логики послужило осознание того, что математика того времени имела шаткое основание и могла приводить к противоречиям. Цель, поставленная создателями современной математической логики, состояла в исправлении этой ситуации и построении прочного и строгого фундамента математики.

Применение логики в программировании привносит некоторые проблемы, часто выходящие за чисто математическое использование логики. Пример, важный для программистской практики, показывает необходимость в некоммутативных вариантах **and** и **or**.

Поставим следующий вопрос, связанный с линией метро *l* и целым числом *n*:

```
"Является ли станция с номером n линии l пересадочной?"
```

Мы могли бы выразить это булевским выражением:

```
l.i_th (n).is_exchange                                    [S1]
```

где *is_exchange* – запрос булевского типа в классе *STATION*, определяющий, является ли станция пересадочной. С запросом *i_th* мы познакомились в предыдущей главе – он возвращает станцию на линии, идентифицируемую по ее номеру, здесь *n*.

Корректная форма этого запроса появится ниже в этой главе.

Выражение **[S1]** появляется для выполнения предназначенной работы: *l* обозначает линию; *l.i_th* (*n*), обозначает ее *n*-ю станцию, экземпляр класса *STATION*; так что *l.i_th* (*n*).*is_exchange*, применяя запрос *is_exchange* к этой станции, говорит нам, используя булевское значение, является ли станция пересадочной.

Но мы ничего не сказали о значении *n*. Потому *l.i_th* (*n*) может быть не определено, поскольку запрос *i_th* имеет предусловие:

```
i_th (i: INTEGER): STATION
        -- i-я станция на этой линии
    require
        not_too_small: i >= 1
        not_too_big: i <= count
```

Как же мы можем написать корректное выражение, сохраняя наши намерения? Если $n < 1$ or $n > l.count$, то разумно полагать, что ответом на наш неформальный вопрос не может быть «Да», так как несуществующая станция не может использоваться для пересадки. Так как в булевском мире может быть всего два ответа, ответом на наш вопрос может быть только «**Нет**!» Формально, булевское выражение должно иметь значение **False**. Для достижения этого поведения мы могли бы попытаться выразить желаемое свойство не в виде**[S1]**, но как

```
(n >= 1) and (n <= count) and l.i_th (n).is_exchange                    [S2]
```

Все еще не корректная форма.

Но и это еще не достаточно хорошо. Проблема в том, что если *n* выходит за свои границы, например, $n = 0$, то последний терм *l.i_th* (*n*).*is_exchange* не определен. Если бы нас интересовало только значение **[S2]**, то можно было бы не беспокоиться. Согласно принципу конъюнкции значение может быть только **False**, поскольку первый терм $n >= 1$ имеет значение **False,** а второй и третий термы не влияют на результат.

Предположим, однако, что выражение появляется в программе и начинает вычисляться в период ее выполнения. Операция **and**, как мы видели, является коммутативной, поэтому вполне законно при выполнении вычислить оба операнда *a* и *b*, а затем комбинировать их значения, используя таблицу истинности для **and**. Но тогда вычисление **[S2]** будет прервано из-за ошибки, возникающей при попытке вычисления последнего терма.

Если бы эти вычисления концептуально были необходимыми, то ничего сделать было бы нельзя. Попытка вычислить выражение с неопределенным значением должна приводить к краху. Это подобно попытке вычислить значение числового выражения 1 / 0. Но во многих случаях предпочтительнее заканчивать вычисление выражения, когда первый терм дает значение **False**, и вместо падения возвращать результат **False**, согласованный с определением **and**.

Это недостижимо для обычных булевских операций: мы не можем предотвратить их компьютерные версии от вычисления обоих операндов, а, следовательно, от риска возникновения ошибки.

Случай, иллюстрируемый данным примером, – вычисление условия, которое имеет смысл, только если выполняется другое условие, – встречается на практике столь часто, что необходимо найти решение этой проблемы. Возможны три способа.

Первый состоит в том, чтобы пытаться исправить ситуацию после возникновения ошибки. Если операнд булевской операции не определен, то вычисление приведет к отказу. Мы

могли бы иметь механизм «захвата» отказов и попытаться посмотреть, достаточно ли других термов для установления значения выражения в целом. Такой механизм означает, что отказ не является настоящей смертью. Он, скорее, напоминает смерть в компьютерных играх, где можно получить новые жизни (пока вы можете платить за них). Такой механизм существует: он называется **обработкой исключений** (**exception handling**) и позволяет вам планировать возможность несчастных случаев и попытки исправления ситуации. Однако в данном случае это было бы (отважимся употребить этот термин) «сверхубийственно» (overkill). Чрезмерно много специальных усилий пришлось бы применять для этой простой и общей ситуации.

Второй путь решения проблемы мог бы заключаться в принятии решения, что операция **and**, как она понимается в программировании, более не является коммутативной (это же верно и для двойственной операции **or**). При вычислении *a* **and** *b* мы бы гарантировали, что *b* не будет вычисляться, если *a* получило значение **False**. Результат в этом случае был бы **False**. Недостаток такого решения в несоответствии компьютерной версии с хорошо продуманной математической концепцией. Есть и прагматический недостаток, связанный с эффективностью вычислений. Дело в том, что коммутативность в случае, когда оба операнда определены, может помочь в более быстром вычислении конъюнкции, вычисляя сначала второй операнд или вычисляя их параллельно, что часто позволяет делать современная аппаратура компьютера.

Такие приемы ускорения вычислений, называемые **оптимизацией**, заботят обычно не программистов, а разработчиков трансляторов.

Третий путь — на котором мы и остановимся — включить дополнительно полезные некоммутативные версии операций, дав им другие имена, во избежание каких-либо семантических недоразумений. Новый вариант для **and** будет называться **and then**; новый вариант для двойственной операции **or** будет называться **or else**. У новых операций имя состоит из двух ключевых слов, разделенных пробелом. Семантика следует из ранее приведенных обоснований.

Почувствуй семантику

Полустрогие булевские операции

Рассмотрим два выражения *a* и *b*, которые могут быть *определены* и иметь булевские значения или могут быть *не определены*. Тогда

- *a* **and then** *b* совпадает со значением *a* **and** *b,* если как *a,* так и *b* определены, и в дополнение имеет значение **False,** если *a* определено и имеет значение **False**;
- *a* **or else** *b* совпадает со значением *a* **or** *b,* если как *a,* так и *b* определены, и в дополнение имеет значение **True**, если *a* определено и имеет значение **True**.

Мы называем новые операции полустрогими, поскольку они строги к первому операнду, требуя его вычисления в любом случае, и снисходительны ко второму, позволяя опускать его вычисление.

Возможно, термин «некоммутативные» был бы более приемлемым, но нам понадобится полустрогий вариант операции, которая изначально не является коммутативной.

Еще один способ определения семантики полустрогих операций состоит в задании расширенной таблицы истинности, где каждый операнд и результат будут иметь три возможных значения: **True**, **False** и *Undefined.*

Всякий раз, когда *a* **and** *b* определено, *a* **and then** *b* также определено и имеет то же значение, но обратное не верно. То же самое верно и для **or else** относительно **or**.

С новой нотацией появляется корректный способ выражения условия нашего примера:

```
((n >= 1) and (n <= count)) and then l.i_th (n).is_exchange        [S3]
```

Предыдущая версия **[S2]** имела две операции **and**, но только вторая из них нуждается в замене на **and then**. Между первыми двумя термами, взятыми в скобки для ясности, обычный **and** достаточно хорош, так как оба операнда будут определены. Дадим общий совет.

Почувствуй методологию

Выбор между обычными и полустрогими операциями

При записи контрактов и других условий:

- используйте обычные булевские операции **or** и **and**, когда можно гарантировать, что оба операнда определены при выполнении программы в тот момент, когда требуется вычислить условие;
- если же гарантировать можно только определенность одного операнда, то делайте его первым и используйте операции **or else** и **and then**.

Наш пример **[S3]** соответствует последнему случаю.

В первом случае не было бы ошибкой использовать полустрогую версию. Но в этом случае предписывался бы строгий порядок вычисления операндов. Стоит избегать излишней сверхспецификации. Это оставляет компиляторам свободу в оптимизации порядка вычисления операндов.

Понятие полустрогой операции применимо не только к математической логике и ПО.

Почувствуй практику

Полуограниченные операции и вы

Полустрогие операции отражают особенности вывода, применимые в повседневной жизни.

Всякий раз, когда вы встречаете фразу «если существует...», можно полагать, что речь идет о полустрогой операции. Кредитные операции могут требовать подписи супруга, если он существует, что можно выразить как *is_single* **or else** *spouse_must_sign* или в явном программистском стиле:

```
(spouse = Void) or else spouse.must_sign
```

где **Void** означает отсутствие объекта. В любой форме второй операнд **or else** не имеет смысла для строгого **or**, так как, когда первый операнд имеет значение **True,** понятие супруга не определено.

Полустрогая импликация

Импликация также имеет полустрогий вариант. Метод с аргументами *l*: *LINE* и *i*: *INTEGER* может использовать предусловие

```
((i >= 1) and (i <= count)) implies l.i_th (i).is_exchange
```

смысл которого состоит в том, что *i*-я станция линии *l*, если существует, является пересадочной.

Этот пример демонстрирует разумность полустрогой интерпретации импликации. Такая схема – выражение в форме *a* **implies** *b*, где *b* определено только тогда, когда *a* истинно, – встречается столь часто, что для этой операции, не являющейся коммутативной, кажется разумным ввести полустрогую версию, подходящую для всех случаев. Это вполне согласуется с принципом импликации и его требованием (предложение I1), что *a* **implies** *b* имеет значение **True**, независимо от *b*, всегда, когда *a* имеет значение **False**.

Поэтому мы выбираем полустрогую версию в качестве определения **implies**.

Определение: импликация с возможными неопределенностями

Значение *a* **implies** *b* для любых *a* и *b,* где *b* может быть неопределенно, является значением

(**not** *a*) **or else** *b*

Пример полустрогости из обычной жизни, цитируемый выше, подпадает под эту категорию. Мы можем теперь записать его с полустрогим **implies** как

```
(spouse /= Void) implies spouse.must_sign
```

Большинство использований «Если существует...», встречающихся в различных документах, следуют этому образцу.

5.4. Исчисление предикатов

Концепции, обсуждаемые до сих пор, принадлежали той части логики, которая называется **исчислением высказываний** (пропозициональным исчислением). Здесь базисными элементами являются *высказывания* (*propositions*), каждое устанавливающее единственное свойство *p*, которое может принимать только два значения – истина и ложь. Вот примеры: «Число *n* положительно», «Я – губернатор Калифорнии», «Сегодня ночью будет полная луна». Единственность свойства в этих примерах означает, что *p* характеризует единственный объект – число, текущую ночь – или конечное множество явно перечисленных объектов, как в высказывании «Я не губернатор, и сегодня ночью нет полной луны».

Другая теория, непосредственно полезная в программах и при их обсуждениях, – **исчисление предикатов**, рассматривающая свойства, характеризующие не отдельные объекты, а множества объектов.

Обобщение «or» и «and»

Пусть дано множество объектов *E* и свойство *p* этих объектов. В исчислении предикатов изучаются два основных вопроса, обобщающих «or» и «and»:

1. Существует ли *по меньшей мере* один объект в *E,* удовлетворяющий *p*?
2. Для каждого из объектов в *E* выполнимо ли *p*?

Например, мы знаем, что линия метро содержит станции, некоторые из которых могут быть пересадочными. Нас могут для конкретной линии интересовать вопросы:

A1. Существует ли (есть ли хоть одна) на Line 8 пересадочная станция?

A2. Все ли станции Line 8 являются пересадочными?

Если вы знаете все станции этой линии по названиям, то эти вопросы можно задать булевскими выражениями. A1 является **or**-выражением и A2 – **and**-выражением:

```
L1  Station_Balard.is_exchange or Station_La_Motte.is_exchange or
    Station_Concorde.is_exchange … [Включить все станции линии] …
L2  Station_Balard.is_exchange and Station_La_Motte.is_exchange and
    Station_Concorde.is_exchange … [Включить все станции линии] …
```

В обоих случаях используется булевский запрос *is_exchange* класса *STATION*, говорящий нам, является ли станция пересадочной. Вы должны включить в выражение терм, задающий каждую станцию.

Можно избежать именования станций, если использовать запрос *i_th* класса *LINE*, который дает нам *i*-ю станцию для любого применимого *i*:

```
M1  Line8.i_th(1) .is_exchange or Line8.i_th(2) .is_exchange or
      … [Включить все целые числа от 1 до Line8.count]
M2  Line8.i_th(1) .is_exchange and Line8.i_th(2) .is_exchange and
      … [Включить все целые числа от 1 до Line8.count]
```

Понятно, что явно перечислять список всех станций весьма неудобно. В частности, нельзя написать выражение, применимое к любой линии, поскольку разные линии имеют разное число станций.

В исчислении предикатов в таких случаях вводятся выражения с **кванторами**, описывающие применение свойств к множеству объектов, позволяя ссылаться только на само множество, например, линию метро, а не на отдельных представителей этого множества – каждую станцию. Вводятся два квантора:

- **квантор существования** – *exists* или ∃ в математической нотации, чтобы установить, что, по крайней мере, один член множества обладает свойством;
- **квантор всеобщности** – *for_all* или ∀ в математической нотации, чтобы установить, что все члены множества обладают свойством.

Когда требуются булевские операции для произвольного числа операндов, *exists* обобщает **or**, а *for_all* обобщает **and**. Если *Line8_stations* обозначает список станций, то математическая нотация позволяет записать:

```
Q1    ∃ s: Line8_stations | s.is_exchange
Q2    ∀ s: Line8_stations | s.is_exchange
```

что можно прочитать соответственно как:

- **существует** *s* в *Line8_stations*, такая, что *s.is_exchang*е истинно;
- **для всех** *s* в *Line8_stations*, *s.is_exchange* истинно.

Для отделения свойства, здесь *s.is_exchange*, от спецификации множества математики часто используют вместо символа вертикальной черты «|» символы «точка» или «запятая». Для нас это неприемлемо, поскольку эти символы уже заняты и играют другую роль в нашем языке.

Выражения Q1 и Q2 представляют математическую, а не программистскую запись. Вскоре мы увидим, как можно выражать эти свойства в программах.

Точное определение: выражение с квантором существования

Нотация, использующая кванторы существования и всеобщности, проиллюстрированная выше, задает новые формы булевских выражений, дополняющих изучаемые ранее в этой главе выражения исчисления высказываний.

Вот определение для квантора существования.

Определение: выражение с квантором существования

Значение выражения

```
∃ s: SOME_SET | s.some_property
```

есть **True**, если и только если по меньшей мере один член заданного множества *SOME_SET* удовлетворяет заданному свойству *some_property*.

Пусть *X* – множество целых чисел{3, 7, 9, 11, 13, 15} (множество, которое состоит из чисел, перечисленных в фигурных скобках). Пусть теперь для любого целого *n* свойство *n.is_odd* означает, что *n* нечетно, свойство *n.is_even* означает, что *n* четно, а *n.is_prime* – что *n* простое число. Тогда

- ∃ *n*: *X* | *n.is_odd* означает, что по крайней мере один из членов *X* является нечетным; выражение имеет значение **True**, так как, например, 3 свидетельствует, что такой член присутствует в множестве. Поскольку все члены нечетны, каждый из них является свидетельством истинности условия;
- ∃ *n*: *X* | *n.is_prime* означает, что по крайней мере один из членов *X* является простым числом; выражение имеет значение **True**, так как, например, 3 свидетельствует, что такой член присутствует в множестве. Не имеет значения, что число 9 — член множества – не является простым числом; нам достаточно хотя бы одного члена множества, для которого выполняется условие;
- ∃ *n*: *X* | *n.is_even* означает, что по крайней мере один из членов *X* является четным; выражение имеет значение **False**, так как в множестве нет четных чисел.

Эти примеры иллюстрируют, как можно доказать или опровергнуть выражение с квантором существования ∃ *s*: *SOME_SET* | *s.some_property*.

E1: Чтобы доказать истинность, достаточно предъявить один элемент множества, удовлетворяющий свойству. Как только такой элемент найден, все остальные члены не влияют на результат. В частности это означает, что нет необходимости в исследовании всех членов множества.

E2: Для опровержения истинности (доказательства ложности) необходимо проверить, что в множестве нет элементов, удовлетворяющих свойству. Недостаточно найти один элемент, для которого свойство не выполняется. В частности это означает, что проверке подлежат **все члены** множества.

Точное определение: выражение с квантором всеобщности

Рассмотрим выражение, использующее квантор всеобщности:

```
∀ s: SOME_SET | s.some_property
```

Его значение равно **True**, если и только если каждый элемент *SOME_SET* удовлетворяет свойству *some_property*. Это не вполне строгое определение из-за возможного случая пустого множества, обсуждаемого ниже. Лучший подход основан на определении, использующем уже известный квантор существования.

Определение: выражение с квантором всеобщности

Значение выражения
$\forall$ *s: SOME_SET* | *s.some_property*
является значением
not($\exists$ *s: SOME_SET* | **not** *s.some_property*)

Отсюда следует, что $\forall$-выражение имеет значение **True,** если и только если нет членов данного множества, которые *не* удовлетворяют данному свойству. На первый взгляд, это определение представляется «перекрученной» версией первого определения. На уроках литературы вам могут сказать, что в разговорах и письменной речи следует избегать двойного отрицания, заменяя фразу: «В этом прекрасном университете нет курсов, которые бы мне не нравились» на фразу «Мне нравятся все курсы в этом университете». Причина двойного отрицания в том, что мы хотим быть аккуратными со случаем пустого множества. Но прежде чем заняться им, давайте снова обратимся к примеру с множеством целых чисел *X*, определенным как{3, 7, 9. 11, 13, 15}:

- $\forall$ *n*: *X* | *n.is_odd* означает, что все члены *X* нечетны; выражение равно **True**, так как 3, 7, 9, 11, 13 и 15 все являются нечетными;
- $\forall$ *n*: *X* | *n.is_prime* означает, что все члены *X* простые числа; выражение равно **False**, так как мы можем взять 9, свидетельствующее, что, по крайней мере, один член множества не является простым числом. Хотя есть и другие члены с таким же свойством, но и одного достаточно для опровержения выражения с квантором всеобщности;
- $\forall$ *n*: *X* | *n.is_even* означает, что все члены *X* – четные числа; выражение равно **False**, так как, например, 3 нечетно. *Любой* член множества мог бы служить для опровержения, поскольку все нечетны, но снова и одного члена достаточно.

Эти примеры иллюстрируют, как можно доказать или опровергнуть выражение с квантором всеобщности $\forall$*s: SOME_SET* | *s.some_property* (сравните с тем, как это делается для выражений с квантором существования, E1 и E2).

U1: Чтобы доказать истинность, следует доказать, что *каждый* элемент *SOME_SET*, если он существует, *удовлетворяет* свойству. То, что некоторые удовлетворяют, не достаточно для доказательства. Это означает, в частности, что следует рассмотреть **все** элементы.

U2: Чтобы опровергнуть истинность (доказать ложность), достаточно предъявить *один* элемент, который *не удовлетворяет* множеству. Как только такой элемент найден, остальные элементы можно не анализировать. Это означает, в частности, что нет необходимости в анализе всех элементов множества.

Отношения между кванторами существования и всеобщности обобщают отношения между **or** и **and**. В частности, следующие два свойства обобщают законы Де Моргана:

```
not (∃ s: E | P) = ∀ s: E | not P
not (∀ s: E | P) = ∃ s: E | not P
```

Первое свойство следует из определения; второе следует из применения первого к **not** *P* и отрицания обеих сторон.

Случай пустых множеств

Множество *SOME_SET*, рассматриваемое в выражениях с кванторами, может быть пустым. Эффект воздействия на него двух кванторов отражает их двойственность.

- ∃ *s*: *SOME_SET* | *s.some_property* истинно в соответствии с определением, если и только если некоторый член *SOME_SET* удовлетворяет *some_property*. Если *SOME_SET* пусто, у него нет членов, и следовательно, нет члена, удовлетворяющего свойству. Потому значение выражения в этом случае независимо от *some_property* всегда есть **False**.
- ∀ *s*: *SOME_SET* | *s.some_property* ложно, если и только если некоторые члены *SOME_SET* *не удовлетворяют* свойству *some_property*. Если *SOME_SET* пусто, то нет члена, играющего роль «контрпримера», поскольку у множества нет членов. Поэтому значения выражения в этом случае всегда есть **True**.

Мы можем также рассматривать второй случай как следствие определения выражения с квантором всеобщности ∀ *s*: *SOME_SET* | *s.some_property*, переходя к квантору существования:

```
not (∃ s: SOME_SET | not s.some_property)
```

По предыдущему соглашению, любое выражение (∃ *s*: *SOME_SET* | ...) в круглых скобках равно **False**, если *SOME_SET* пусто. Потому ∀-выражение, выводимое с применением **not**, имеет значение **True**.

На практике это означает, что можно полагать истинным каждое предложение в форме «Каждый объект такого или такого вида удовлетворяет любому свойству», если только нет объектов указанного вида. Так что можно пообещать студентам: «Я обещаю вам, что каждый блондин в этой аудитории станет губернатором еще до конца этого года, а если нет, то я каждому из вас выплачу по миллиону долларов (буду ставить только отличные оценки)». Конечно, такое обещание можно дать, если предварительно вы тщательно проверили и убедились, что у всех студентов в аудитории есть черные волосы. Тогда блондинов нет и утверждение, что блондины станут губернаторами, истинно.

Несмотря на изучение логики, никогда не следует обещать ничего подобного: «Каждый студент блондин из этой аудитории станет губернатором», поскольку вы берете на себя ответственность за идентификацию крашеных блондинов, а *также и* за возможную фальсификацию выборов.

Как следствие этих наблюдений, официальное имя для квантора всеобщности «*For all*» мы не можем признать вполне удачным, поскольку неформально «Для всех» предполагает существование некоторых элементов, о которых идет речь. Лучшим именем было бы «*For all, if any*» или просто «*For any*».

(Было бы прекрасно называть два квантора симметрично: «*For some*» и «*For any*» – «Для некоторого» и «Для любого»)

Смена устоявшихся терминов привела бы к совершенно ненужной путанице, поэтому по-прежнему будем говорить «*For all*» и «*Exists*», но следует помнить, что это только формальные имена и математическая интерпретация *For all* дает **True**, так же как и *Exists* дает **False**, если *нет элементов*, для которых можно было бы проверить выполнимость условия.

Другой способ выражения этого свойства состоит в представлении квантора существования записью a_1 **or** a_2 **or** … **or** a_n, а квантора всеобщности – a_1 **and** a_2 **and** … **and** a_n. Тогда, если *n* равно нулю, дизъюнкция ложна, а конъюнкция истинна. Последнее следует из определения операций: *a* **or** *b* истинно, если и только если по меньшей мере один из элементов *a* или *b* истинен, и *a* **and** *b* ложно, если и только если по крайней мере один из элементов *a* или *b* ложен.

Еще одна неформальная интерпретация связывает это свойство с предыдущим обсуждением импликации, когда ложная посылка всегда дает истинность импликации. Мы можем понимать ∀ *x: SOME_SET* | *x.some_property* как следующую импликацию: «*x* is a member of *SOME_SET*» implies *x.some_property*.
Если *SOME_SET* пусто, то посылка ложна для каждого возможного x, поэтому импликация истинна.

5.5. Дальнейшее чтение

Материал этой главы является введением, частью учебного плана по информатике, так что большинство студентов будет иметь отдельный курс, специально посвященный логике. Стандартным учебником, требующим определенной математической подготовки, является

Elliot Mendelson: *Introduction to Mathematical Logic*, fourth edition, Chapman & Hall/CRC, 1997. (Э. Мендельсон «Введение в математическую логику», Наука, М, 1976 г.)

Следующие учебники обращены непосредственно к тем, кто изучает информатику:

Zohar Manna, Richard Waldinger: *The Deductive Foundations of Computer Programming*, Addison-Wesley, 1993.

Mordechai Ben-Ari: *Mathematical Logic for Computer Science*, 2nd edition (2001), Springer-Verlag, third corrected printing, 2008.

5.6. Ключевые концепции, изучаемые в этой главе

- Логика точно и строго определяет способы вывода. Она обеспечивает основу как для математики, так и для программирования.
- Пропозициональное исчисление определяет операции над «булевскими переменными», которые могут принимать только значения **True** и **False**. Базисными «булевскими операциями» являются отрицание (**not**), дизъюнкция (**or**) и конъюнкция (**and**).
- Дизъюнкция и конъюнкция являются двойственными операциями по отношению друг к другу. Отрицание одной операции, примененное к отрицанию операндов, дает другую операцию. Законы Де Моргана отражают этот факт.
- Дизъюнкция и конъюнкция могут быть обобщены на произвольное число операндов благодаря введению кванторов ∃ и ∀ – существования и всеобщности исчисления предикатов, которые применяются к членам заданного множества.
- Для пустого множества вне зависимости от проверяемого свойства квантор существования дает значение **False**, а квантор всеобщности – значение **True**.
- Импликация может быть просто выражена через дизъюнкцию и отрицание: *a* **implies** *b* – то же самое, что и (**not** *a*) **or** *b*. Импликация может использоваться для вывода новых свойств из ранее доказанных. Она не подразумевает причинности. **False** влечет **True**.
- В приложениях к программированию булевские операции имеют полустрогие версии, вырабатывающие значения даже в тех случаях, в которых второй операнд не определен. Этими полустрогими версиями для **or** и **and** являются **or else** и **and then**. Для импликации **implies** лучшее определение дает полустрогая версия.

Новый словарь

Antecedent	Посылка	**Boolean value**	Булевское значение
Boolean expression	Булевское выражение	**Boolean operator**	Булевская операция
Boolean variable	Булевская переменная	**Conjunction**	Конъюнкция
Consequent	Заключение (следствие)	**Contradiction**	Противоречие
Disjunction	Дизъюнкция	**Existential quantifier**	Квантор существования
Implication	Импликация	**Logic**	Логика
Negation	Отрицание	**Opposite**	Противополож-ность
Predicate calculus	Исчисление предикатов	**Propositional calculus**	Пропозицио-нальное исчисле-ние (исчисление высказываний)
Quantifier	Квантор		
Satisfiable	Выполнимое		
Satisfies	Удовлетворяет		
Strict	Строгий	**Stronger**	Сильнее
Tautology	Тавтология	**Truth assignment**	Истинностное высказывание (распределение)
Truth table	Таблица истинностино стная таблица) (истин		
Universal quantifier	Квантор всеобщности	**Weaker**	Слабее

5-У. Упражнения

5-У.1. Словарь

Дайте точные определения всех терминов словаря.

5-У.2. Карта концепций

1. Создайте карту концепций для терминов словаря.
2. Скомбинируйте эту карту с картой, составленной в предыдущих главах.

5-У.3. Свойства булевских операций

(Обоснуйте ваши ответы)

1. Обладает ли свойством рефлексивности **and**?
2. Обладает ли свойством рефлексивности **or**?
3. Обладает ли свойством ассоциативности эквивалентность?

5-У.4. Искаженная логика

«*Если температура в городе поднимается выше 30 градусов, то возникает угроза загрязнений. Сегодня температура достигла только 28 градусов, поэтому угрозы загрязнений нет*».

1. Неформально: что ошибочно в этом высказывании?
2. Введите подходящие булевские переменные, представив это высказывание в виде булевского выражения.
3. Докажите, что оно не является тавтологией (подсказка: постройте истинностное присваивание, при котором выражение становится ложным).
4. Является ли оно противоречием? (Обоснуйте ответ)

5-У.5. Соответствует ли предупреждение?

На входе в компьютерный центр можно прочитать: «Вход запрещен всем, кто не авторизован или не имеет сопровождающего». Можно считать, что здесь нет неопределенности и авторизованный – это человек, обладающий нужными правами, а сопровождающим является авторизованный человек.

1. Введите подходящие булевские переменные и выразите правило на входе в виде булевского выражения.
2. Объясните, почему правило включает случай запрета, которого, видимо, не добивался автор объявления (подсказка: используйте законы Де Моргана).
3. Запишите выражение, соответствующее намерениям автора объявления.
4. Используя выражение как руководство, перепишите текст объявления.

5-У.6. Неравенство

Выпишите таблицу истинности для неравенства /=.

5-У.7. Ассоциативность и импликация

Является ли импликация **implies** ассоциативной? (Обоснуйте ответ)

5-У.8. Признаки силы

Мы говорим, что *a* «сильнее или равно» *b,* если *a* implies *b*.

Докажите, что *a* **and** *b* сильнее или равно *a* и что *a* сильнее или равно *a* **or** *b*.

5-У.9. Импликация и отрицание

При обсуждении импликации отмечалось, что

```
(a implies b) = ((not a) implies (not b))
```

не является тавтологией. Упростив это выражение, используя приведенные в этой главе теоремы и не используя таблицу истинности, покажите, при каких условиях (приведите истинностное присваивание) оно выполняется.

5-У.10. Импликация

1. Докажите, что для любых булевских выражений *a* и *b* следующее выражение является тавтологией:

((*a* **implies** *b*) **and** ((**not** *a*) **implies** *b*)) **implies** *b*

2. На дорожном знаке, установленном в Цюрихе вблизи ETH, написано: «*Разумные водители здесь не паркуются. Остальным парковка запрещена*». Используя подходящие булевские переменные, включающие: *is_reasonable*, *parks_here*, *parking_prohibited*, выразите это предписание в виде булевского выражения.
3. Докажите, что если это выражение является тавтологией и водители выполняют предписание, то *parks_here* ложно.

5-У.11. «Исключающее или» как начало всех булевских операций

Булевская операция «исключающее или», **xor**, является истинной, если и только если истинно либо *a*, либо *b*, но не оба вместе. Мы можем установить это свойство, определив *a* **xor** *b* как

```
(a or b) and (not (a and b))                                        [X1]
```

1. Напишите таблицу истинности для **xor**.
2. Если *a* – булевская переменная, то что является значением выражения *a* **xor** *a*? (Обоснуйте ваш ответ либо построением таблицы истинности, либо как следствие определения)
3. Докажите, что *a* **xor** *b* всегда имеет то же значение, что и **not** (*a* = *b*).

Для каждого из последующих булевских выражений (с нулем, одним или двумя операндами) постройте другое булевское выражение, которое для любого значения операндов дает тот же ответ, что и данное выражение, и использует только заданные операнды, константу **True** и операцию **xor** (другие операции не используются). Ответы обоснуйте.

4. **False**
5. **not** *a*
6. *a* = *b*
7. *a* **and** *b*
8. *a* **or** *b*
9. *a* **implies** *b*

Возможность выразить через **xor** любую булевскую операцию делает эту операцию особенно интересной. Ее можно рассматривать как основу при построении других операций, что и используют проектировщики электронных цепей, основанных на булевской логике.

5-У.12. Свойства «исключающего или»

Основываясь на определении [X1] для **xor,** докажите или опровергните следующие свойства:

1. Операция xor коммутативна.
2. Операция xor ассоциативна.
3. Для любых *a* и *x* имеет место *x* **xor** (*a* **xor** *x*) = *x*.

5-У.13. Голубые шляпы и красные шляпы

Сто человек выстроены в ряд, на каждом одета шляпа, красная или голубая. Каждый может видеть цвет шляпы тех, кто стоит впереди, но не видит цвет шляп стоящих сзади и не знает цвета своей шляпы.

Начиная с конца ряда, с человека, который видит цвета шляп всех, кроме себя, каждый по очереди выкрикивает одно из двух – **«красный»** или **«голубой»**, и это слышат все в ряду.

Придумайте стратегию, при которой как можно большее число людей будут гарантированно правильно называть цвет собственной шляпы. О распределении цветов ничего не известно, и стратегия не основывается на вероятностях правильного ответа. Стратегия учитывает только гарантировано правильные ответы.

Следующие замечания могут быть полезными.

- Простая стратегия заключается в том, что нечетные номера (предполагается, что нумерация начинается с последнего в ряду) выкрикивают цвет четных номеров, стоящих перед ними. Четные номера выкрикивают цвет своей шляпы, названный предшественником. Эта стратегия гарантирует, что половина людей в ряду правильно назовут цвет своей шляпы (фактически, правильных ответов будет больше; при равномерном распределении цветов вероятность правильных ответов для такой стратегии равна 0,75).
- Другая стратегия, гарантирующая те же 50 процентов правильных ответов, состоит в том, что последний в ряду, тот, кто первым дает ответ, называет цвет шляп, преоблада-

ющий в ряду, – «красный», если в ряду 50 или более красных шляп, и «синий» в противном случае. Все остальные повторяют цвет, названный первым.

- Нет стратегии, гарантирующей 100 процентов правильных ответов, так как цвет шляпы последнего в строю никто не знает, так что оценка сверху для оптимальной стратегии равна 99 из 100. Как близко можно подойти к оптимальной стратегии?
- Еще раз напомним, что решение не должно оперировать с вероятностями. Стратегия должна максимизировать число гарантированно корректных ответов.

Подсказка: Может помочь предыдущее упражнение[1].

5-У.14. Истинностные таблицы с неопределенностями: полустрогие булевские операции

Расширим исчисление высказываний, допуская три значения вместо двух: **True**, **False**, *Undefined*. Например, запрос *l*•*i_th* (*i*) будет иметь значение *Undefined*, если он не удовлетворяет предусловию.

Полагая, что *a*, *b* и результирующее выражение может принимать любое из трех значений, постройте таблицу истинности, содержащую 9 строк для

1) *a* **or else** *b;*
2) |*a* **and then** *b*.

5-У.15. Таблица истинности с неопределенностями: обычные булевские операции

Как и в предыдущем упражнении, предположим, что булевские значения включают **True**, **False** и *Undefined*. Обоснуйте ваше решение при построении таблицы истинности для:

1) |*a* **or** *b;*
2) |*a* **and** *b*.

Смысл могут иметь несколько истинностных таблиц, а потому интересно обоснование предлагаемого вами выбора.

[1] Красивая задача, допускающая естественное обобщение. Если вы нашли решение для двух цветов шляп, то попробуйте решить задачу для трех цветов, а потом для *n* цветов шляп

6 Создание объектов и выполняемых систем

После экскурсии в математические основания вернемся к технике программирования.

В предыдущих примерах использовались имена, такие как *Paris* и *Route1*, для получения доступа к объектам, которые кем-то были созданы для нас, – и все это было довольно загадочно. Настало время разобраться, как самим создавать свои собственные объекты. Создание объектов – центральная тема этой главы – представляет интересный механизм с важными следствиями. Это рассмотрение приведет нас к общей картине выполнения систем: как программа запускается, как она выполняется и как завершается.

В процессе изучения мы создадим фиктивную линию метро, *fancy_line*, связывающую несколько реальных станций. В противоположность предыдущим примерам, таким как *Line8*, новая линия *fancy_line* не предопределена, мы должны построить ее сами. Этот процесс потребует создания других объектов, например, для представления остановок на линии.

Три линии парижского метро завершаются на южной стороне примерно в одном районе, но довольно далеко для пешеходных переходов. Жителям этого района для перехода с одной линии на другую приходится ехать в центр, там выполнять пересадку на другую линию. Цель *fancy_line* – облегчить их участь, создав кольцевую линию, если и не в реальном городе, то хотя бы в нашем виртуальном мире.

Рис. 6.1.

6.1. Общие установки

Наша система для этой главы называется *creation*. Откройте ее теперь, используя те же приемы, что и в предыдущих главах. Перейдите к классу, предмету данного обсуждения, *LINE_BUILDING,* который изначально выглядит так:

```
class LINE_BUILDING inherit
      TOURISM
feature
               build_a_line
                                 -- Построить воображаемую линию и подсветить ее на карте.
                                 do
                                          Paris.display
           -- "Создать новую линию, заполнить ее станциями и добавить в Paris"
               -- "Подсветить новую линию на карте"
                                 end
end
```

Строка —«Создать новую линию, заполнить ее станциями и добавить в *Paris*» и строка, следующая за ней, начинаются с двух дефисов и, следовательно, являются комментариями. Но это комментарии особого рода, известные как **псевдокод**, означающие, что они замещают реальный программный текст – код, который мы собираемся поместить в этом месте, разрабатывая программу.

Определение: Псевдокод

Псевдокод является неформальным текстом, замещающим программные элементы, которые будут добавлены позднее.

При разработке программы полезно использовать псевдокод, это лучше, чем вместо программного элемента писать комментарий: «Сюда следует поместить программный код». Псевдокод позволяет дать неформальное описание будущего кода, который пока мы не готовы поместить, например, из-за того, что это потребовало погружения в детали, мешающие увидеть всю картину в целом. На момент написания псевдокода детали могут быть еще не продуманы.

Процесс постепенной замены псевдокода элементами фактического кода называется **детализацией,** или **уточнением (refinement)**. Этот прием еще более полезен при написании сложного ПО. Он является частью *проектирования сверху вниз*, обсуждаемого в последней главе.

Псевдокод будет использовать соглашения, иллюстрируемые примером.

Почувствуй стиль: Запись псевдокода

Записывайте элементы псевдокода как комментарии, заключая их текст в кавычки.

После добавления псевдокода следует убедиться, что наша программа, даже если она и не полна, синтаксически корректна. Возможно, выполнять ее не имеет смысла, но компилироваться она должна, чтобы компилятор мог обнаружить ошибки, проскользнувшие по недосмотру, например, некорректное использование типов. Это базисное правило методологии разработки: **программа должна компилироваться на любом этапе ее разработки**. Служебные программы, рассматриваемые в последней главе, обеспечивают дополнительные способы (обычно превосходящие псевдокод), направленные на достижение этой цели.

Создание комментируемого псевдокода напоминает, что это не обычный комментарий, аннотирующий существующий код, а держатель места для кода, который в конечном итоге должен появиться в этом месте.

6.2. Сущности и объекты

Наш класс нуждается в компоненте (запросе), представляющем линию, которую мы собираемся построить. Мы назовем этот запрос *fancy_line*. Это даст возможность начать процесс детализации, превращая часть псевдокода (часть первой строки и полностью вторую) в фактический код и делая напоминание более точным:

```
class LINE_BUILDING inherit
  TOURISM
feature
    build_a_line
                          – Построить воображаемую линию и подсветить ее на карте.
                do
                          Paris.display
                          – "Создать fancy_line, заполнить ее станциями"
                          Paris.put_line (fancy_line)
                          fancy_line.highlight
                end
    fancy_line: LINE
                          – Воображаемая (но желательная) линия Парижского метро
end
```

Рассмотрим строчку, следующую за псевдокодом. Оператор «*Paris.put_line* (*fancy_line*)» заменил текст псевдокода «добавить в *Paris*». Команда *put_line*, доступная для объектов класса *CITY*, позволяет добавить в «город» линию со станциями. Следующий за ним оператор использует команду *highlight* для уточнения второй строки исходного псевдокода.

Сразу же после выполнения процедуры *build_a_line* идентификатор *fancy_line* будет обозначать экземпляр класса *LINE*, задающий линию метро.

Идентификаторы могут обозначать многие вещи: они могут быть именами классов, подобно *STATION*, или методов, подобно *i_th*. Идентификатор, такой как *fancy_line*, чья роль – обозначать значение объекта, существующего в период выполнения, называется **сущностью**.

Если у вас есть опыт программирования, вы должны быть знакомы с еще одним важным понятием – понятием «переменной», обозначающим сущность, значение которой может изменяться. «Сущность» – более общее понятие, поскольку она может иметь и константное значение. В последующих главах переменные будут изучаться в деталях.

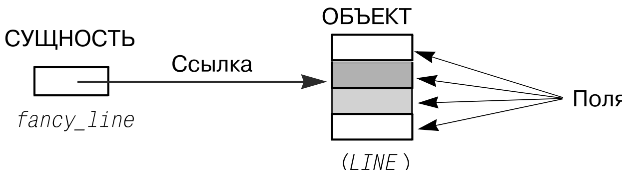


Рис. 6.2. В период выполнения: сущность и связанный с ней объект

В данном случае сущность *fancy_line* является именем атрибута, но нам придется встречаться и с другими видами сущностей.

Если, в некоторый момент выполнения сущность представляет объект, мы будем говорить, что сущность **присоединена** к объекту

Следующий рисунок, изображающий момент выполнения, помогает визуализировать понятие сущности и присоединенного объекта.

Детализируем существующие отношения.

- Сущность – это имя в программе, которое в момент выполнения будет обозначать, благодаря «ссылке», объект в памяти. Понятие ссылки, выражающее связь, будет более точно определено в последующих главах.
- Объект, как определено ранее, является коллекцией данных или, если говорить более точно, как показано на рисунке, он состоит из множества **полей**, каждое из которых содержит единицу данных (например, целое или булевское значение). Данные, которыми наша программа манипулирует во время выполнения, полностью созданы из таких объектов, каждый со своим набором полей. Поля объекта *STATION* могут, например, включать координаты станции на карте, ее имя и другие характеристики.

Соглашения по изображению диаграмм, дающих мгновенный снимок структуры объекта или его части во время выполнения, следующие:

- объект представляется прямоугольником с внутренними прямоугольниками, задающими поля объекта;
- рядом с каждым объектом, обычно ниже, приводится в скобках имя класса этого объекта (на рисунке *LINE*).

6.3. VOID-ссылки

При рассмотрении выполнения *build_a_line* и значения *fancy_line* особое внимание следует обратить на ссылки и их связи с объектами.

Начальное состояние ссылки

Предположим, что у нас есть экземпляр класса *LINE_BUILDING*. Возможно, вы думаете, что поскольку класс объявил запрос *fancy_line* типа *LINE*, отсюда всегда следует, что этот экземпляр содержит ссылку на экземпляр *LINE*, как показано на рисунке:

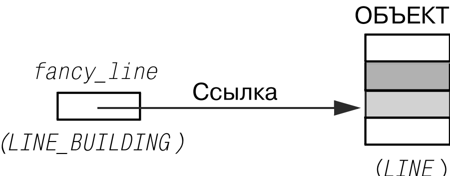


Рис. 6.3. Объекты LINE_BUILDING и LINE

Это не так. У нас есть объект, показанный слева на рисунке, – экземпляр *LINE_BUILDING*, с одним полем, соответствующим запросу *fancy_line*. Давайте предположим, что это объект уже создан в результате вызова «оператора создания», который мы вскоре научимся писать. Этот оператор даст нам только один объект: *LINE_BUILDING*. Если нам нужен дру-

гой объект, то он должен быть явно создан в программе, потому состояние программы сразу после создания экземпляра *LINE_BUILDING* выглядит так:

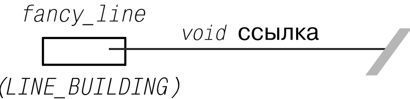


Рис. 6.4. Структура объекта в начале выполнения

Поле *fancy_line* содержит ссылку. Но поскольку все же нет оператора, создающего другие объекты, ссылка имеет значение **void**, означающее, что она не присоединена ни к какому объекту. Рисунок отражает принятое соглашение для изображения void-ссылок, напоминая «заземление» в электрических цепях.

Ссылка может находиться в одном из двух возможных состояний.

Определение: состояния ссылки

В любой момент выполнения значение сущности, обозначающее ссылку, может быть:

- **присоединенным** к некоторому объекту;
- **void**.

Предопределенный компонент **Void** обозначает ссылку. Так что в любой момент выполнения, если *x* обозначает ссылку, условие

```
x = Void
```

имеет значение *True*, если и только если значение *x* является void-ссылкой; и

```
x /= Void
```

если и только если сущность *x* присоединена к объекту.

Трудности с void-ссылками

Базисный механизм вычисления был представлен как *вызов метода* в форме *x.f* или с аргументами *x.f* (...). Метод применяется к объекту, к которому *x* присоединен. Но теперь, при наличии ссылок, возникает возможность, что в некоторый момент выполнения, если верно *x* = **Void**, ссылка, которую обозначает *x*, не присоединена ни к какому объекту. Вызов метода в этом случае будет ошибочным.

Чтобы увидеть эффект от такого «жучка», попытаемся выполнить систему в следующей форме:

```
class LINE_BUILDING inherit
    TOURISM
feature
        build_a_line
                -- Построить воображаемую линию и подсветить ее на карте.
```

```
            do
               Paris .display
 -          Paris .put_line (fancy_line)
                        - Следующая строка должна быть заменена кодом!
                  - "Создать fancy_line и подсветить ее на карте"
                  fancy_line .highlight
            end
         fancy_line: LINE
  end
```

Как показано, следует временно закомментировать строку *Paris.put_line* (*fancy_line*). Эта строка сама по себе содержит ошибку, но нас сейчас интересует другая ошибка.

После начального вызова (*Paris.display*) выполнение будет прервано, и появится сообщение, указывающее, что встретилось **исключение**.

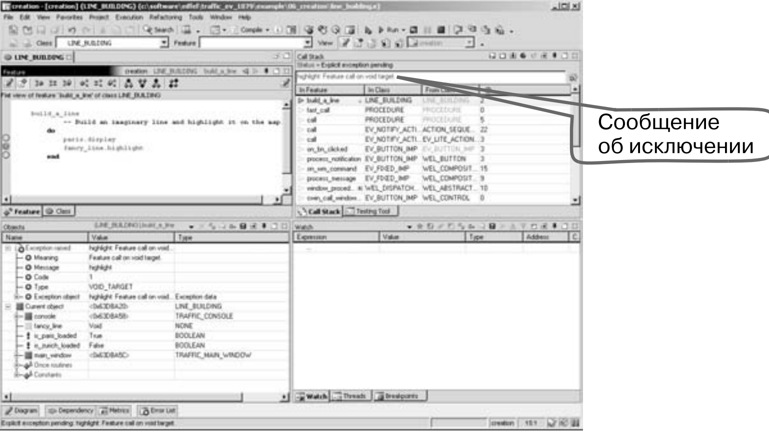


Рис. 6.5. Завершение по прерыванию

То, что случилось, является void-вызовом, или попыткой вызова метода void-целью, – вызовом метода несуществующим объектом. Такая попытка никогда не может быть успешной.

Почувствуй Семантику: Принцип присоединенной цели

> Вызов в форме *x.f* (...) может выполняться нужным образом только при условии, что в момент его выполнения *x* является присоединенным.

При void-вызове возникает *исключение*: событие, вызывающее прерывание нормального процесса выполнения программы. В данном случае исключение привело к отказу выполнения всей программы. Сообщение *EiffelStudio* указывает имя происшедшего исключения: «Feature call on void reference» (вызов метода void-ссылкой).

Изучая исключения, мы увидим, что можно избежать отказа, подготовив код обработки исключения, который будет пытаться восстановить ситуацию. Но это приемы «последней надежды». Предотвратить – лучше, чем лечить. Общее правило: всякий раз, когда в про-

грамме встречается вызов *x.f* (...), при котором возможно, что *x* может быть void, защитите вызов, чтобы он мог выполняться только тогда, когда *x* присоединен. Примером является библиотечный код, где можно увидеть много проверок вида **if** *x* /= **Void then ...** и предусловий в форме *x* /= **Void**, когда *x* – это аргумент вызова. Вы должны проследить, чтобы цели всех вызовов были присоединенными при каждом выполнении.

Ситуация улучшается. Приложение к этой главе описывает современную эволюцию, при которой полностью исключается риск void-вызовов.

Если раскомментировать строку *Paris.put_line* (*fancy_line*), перекомпилировать и выполнить программу, то произойдет отказ в работе, но по другой причине – нарушено предусловие метода *put_line*, которое устанавливает, что аргумент не должен быть void (причина того, что эту строку следовало вначале закомментировать, состояла в том, что хотелось вначале познакомиться с ситуацией void-вызова).

Не каждое объявление должно создавать объект

Чтобы избежать void-вызова и возникновения исключения в последнем примере, мы можем изменить процедуру создания *build_a_line* так, чтобы перед вызовом *fancy_line.highlight* объект уже был создан и присоединен к *fancy_line*. Вскоре мы это и сделаем. Но стоит задаться вопросом, зачем вообще нужны void-ссылки? Не следует ли полагать, что объявление, такое как

```
fancy_line: LINE
```

при выполнении будет иметь эффект создания объекта – экземпляра *LINE* – и присоединения его к *fancy_line*?

Ответ – нет. Есть несколько причин, по которым в момент создания ссылки инициализируются значениями void и требуется явное создание объектов в вашей программе.

Основная причина в том, что некоторые объекты просто не существуют. Это справедливо и в обычном мире. Человек *может* иметь супруга, но не у каждого он есть. ПО, моделирующее наш мир, должно учитывать такие свойства. В классе *PERSON*, появляющемся в ПО, управляющем налогами, вероятно, будет создан компонент, идентифицирующий супруга,

```
spouse: PERSON
```

для которого полезна void-ссылка, представляющая случай отсутствия супруга. Даже если предположить, что все женаты и все замужем, то и в этом случае не имеет смысла создавать объект *spouse* каждый раз, когда создается объект *PERSON*. Такая попытка привела бы к бесконечной череде создания объектов, поскольку каждый супруг является экземпляром класса *PERSON* и, следовательно, имеет поле *spouse,* которое является объектом класса *PERSON*, которое... и так далее.

Поэтому разумное решение состоит в том, чтобы инициализировать объекты значением void и создавать объекты явно в том момент, когда они понадобятся или появляются естественным образом.

Рассмотрим наш пример детальнее. Когда человек имеет супруга, то возникает ограничение взаимной пары: у супруга есть супруг, и тогда ссылки должны быть зависимыми –картинка показывает это лучше слов:

Рис. 6.6. Моногамия

Формула говорит это даже лучше, чем рисунок: инвариант класса *PERSON* должен включать предложение

```
monogamy: (spouse /= Void) implies (spouse.spouse = Current)
```

Ключевое слово **Current** обозначает текущий объект, формальное определение которого будет дано чуть позже в этой главе. Инвариант класса содержательно говорит, что если некто имеет супруга, то супруг супруга является тем самым некто.

Так как **Current** обозначает текущий уже существующий объект, то он никогда не может быть void. Отсюда, в частности, следует: (*spouse* /= **Void**) **implies** (*spouse.spouse* /= **Void**). Если вы замужем, то ваш муж женат. Не следует недооценивать преимущества таких, казалось бы, банальностей. Разработка ПО включает прояснение интуитивных знаний о проблемной области и их формализацию, например, в инвариантах класса.

Еще одно наблюдение о вышеприведенном инварианте класса: если вы внимательно следили за обсуждением полустрогих булевских операций, то должны заметить, что этот инвариант требует полустрогой версии импликации, так как второй операнд не будет определен для *spouse,* имеющего значение **Void**.

Все это обсуждение показывает, почему не следует немедленно создавать объект после объявления сущности. Оба объекта на предыдущем рисунке должны начинать свою жизнь самостоятельно, не вступая до поры до времени в брак, сохраняя свои ссылки *spouse* как void.

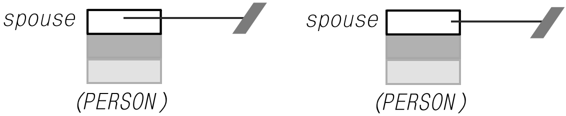


Рис. 6.7. Двойное безбрачие

Позднее, при выполнении, например, некоторой команды *marry*, произойдет взаимное присоединение *spouse*-ссылок, и объекты будут присоединены друг к другу, переходя в состояние, показанное на предыдущем рисунке. Такие команды взаимного присоединения не создают новых объектов – они просто присоединяют ссылки к существующим объектам.

Роль void-ссылок

Рассмотрим ссылку, появляющуюся в поле объекта, такую как поле *spouse* объекта *person*. Если ссылка присоединена к объекту, то это указывает на присутствие некоторой информации, представленной этим объектом. Если же ссылка – void, то это означает, что информация не существует. В частности, это крайне полезно при связывании объектов в сложные структу-

ры. Многие интересные структуры данных, например, *связные списки*, играющие важнейшую роль в наших обсуждениях, основаны на этой концепции связывания.

Рассмотрим простой пример из Traffic. Мы можем решить представлять линию метро (любой экземпляр класса *LINE*) одним или несколькими экземплярами класса *STOP*, каждый из которых представляет остановку на линии. Один из возможных приемов (увидим и другие) состоит в том, чтобы у каждого экземпляра *STOP* существовало поле *right*, указывающее на следующую остановку. Так что экземпляр класса *STOP* может выглядеть примерно так:

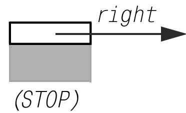


Рис. 6.8. Stop (временно)

где затушеванная часть представляет поля, обеспечивающие другую информацию об остановках. Тогда линия метро будет представлена множеством таких объектов, каждый из которых, кроме последнего, связан со следующим ссылкой *right*:

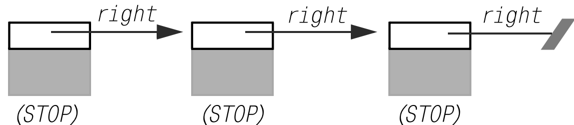


Рис. 6.9. Связанная линия

Заметьте, как последний объект использует void-ссылку, чтобы показать, что у него нет *right*-объекта справа. Завершение таких структур – одно из принципиальных использований ссылок void.

Имя *right* для поля, содержащего ссылку на следующий объект, идет от стандартного способа изображения таких структур с элементами, следующими друг за другом слева направо.

Вызовы в выражениях: помогут победить ваш страх перед void

До того как вернуться к созданию объектов, – что позволяет создавать не-void-ссылки, – следует взглянуть на общую схему использования ссылок в выражениях и попытаться избежать любого страха перед void-вызовами.

Поскольку вызов метода определен только для не-void-цели, то можно задуматься: а как выразить условие, чтобы оно всегда было определено, даже если оно включает вызов? Условному оператору, так же как и инварианту класса, может потребоваться условие в форме

```
fancy_line.count >= 6
```

Условие говорит, что у *fancy_line* по меньшей мере 6 станций. Но это только в том случае, если линия *fancy_line* присоединена к объекту. Если уверенности нет, то необходим способ выражения в неформальных терминах, а это верно, если и только если

```
"fancy_line определена, and then имеет по меньшей мере 6 станций"
```

Вы уже знаете решение (я надеюсь, что «*and then*» послужило звонком): полустрогие операции спроектированы для условий, в которых одна часть имеет смысл, только если другая часть имеет значение **True** (для **and then**) или **False** (для **or else**).

Теперь можно корректно записать это условие как

```
(fancy_line /= Void) and then (fancy_line.count >= 6)
```

Это гарантирует, что условие всегда определено:

- если *fancy_line* имеет значение void, результатом является *False*. Вычисление выражения не будет использовать второй операнд, который стал бы причиной исключения;
- если *fancy_line* присоединена, то второй операнд определен и будет вычислен, что даст окончательное значение результата выражения.

Другой вариант этого образца использует импликацию **implies**, которая определена как полустрогая операция (наряду с **and then** и **or else**). Условие в форме

```
(fancy_line /= Void) implies (fancy_line.count >= 6)
```

выражает слегка отличное условие:

```
"Если fancy_line определена, то следует, что у нее по меньшей мере 6 станций"
```

В отличие от **and then**, дающей **False**, импликация говорит, что «если не определено, то тоже ничего страшного», и дает значение **True**, когда первый операнд ложен. Такой образец часто полезен в инвариантах класса. Он встречался при рассмотрении класса *PERSON*:

```
monogamy: (spouse /= Void) implies (spouse.spouse = Current)
```

Из условия следует, что если вы замужем (женаты), то супруг супруга – это вы сами. Но если супруга нет, то результат все равно **True**. В противном случае, холостяки нарушали бы инвариант, к чему нет никаких оснований. Операция импликации **implies** корректно работает в этой ситуации, так как **False** влечет **True**: ее полустрогость гарантирует отсутствие прерываний при вычислении условия во всех случаях.

6.4. Создание простых объектов

Я надеюсь, вы еще не забыли цель этой главы – создать новую линию *fancy_line* с тремя станциями, как показано на рисунке в начале текста. Мы уже почти приблизились к цели, но прежде нам нужно создать объекты, представляющие остановки на линии.

Эти вспомогательные объекты будут экземплярами уже упомянутого класса *STOP*. Кстати, понимаете ли вы, почему нам нужен такой класс?

Время теста

Остановки на линии метро – это нечто большее, чем станции метро.
Почему нам нужен класс *STOP* для моделирования линии метро и почему недостаточно класса *STATION*?

Следующий ниже рисунок дает подсказку. Остановка связана со станцией, но является другим объектом, поскольку представляет станцию, *принадлежащую определенной линии*.

Запрос «Какая следующая остановка?» не является методом станции (*STATION*) – это метод станции, принадлежащей линии (*STOP*). Вспомним документ требований: «Некоторые станции принадлежат двум или более линиям; они называются пересадочными». На следующем рисунке станция, следующая за станцией «Gambetta», зависит от того, на какой линии вы находитесь.

Рис. 6.10. Более одной «следующей» станции

Объект *STOP* устроен просто. Он содержит ссылки на станцию, на линию и на следующую остановку:

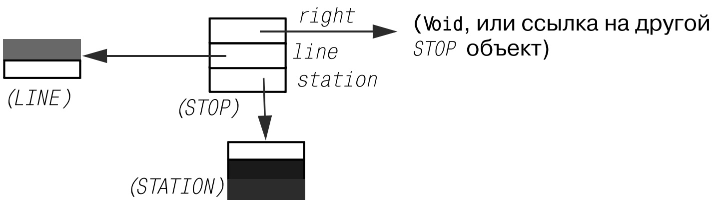


Рис. 6.11. Остановка (Stop) финальная версия

Не имеет смысла иметь остановку без станции и линии. Поэтому будем требовать, чтобы *station* и *line* всегда были присоединены (не void); инвариант класса будет отражать это требование. Ссылка *right* может иметь значение void, чтобы указать, что данная остановка является последней на линии.

Нам нет необходимости заботиться о создании объектов *STATION*, поскольку они предопределены в классе *TOURISM* и доступны через *Station_X*-запросы: *Station_Montrouge*, *Station_Issy* и другие. Поэтому будем учиться созданию объектов на примере создания экземпляров класса *STOP*.

Первую версию класса *STOP* назовем *SIMPLE_STOP*, она имеет следующий интерфейс (доступный для просмотра в EiffelStudio):

```
class SIMPLE_STOP feature
    station: STATION
            -- Станция, которую представляет эта остановка
    line: LINE
```

```
            — Линия, на которой находится эта остановка
    right: SIMPLE_STOP
            — Следующая остановка на той же линии.
    set_station_and_line (s: STATION; l: LINE)
            — Связать эту остановку с s и l.
        require
            station_exists: s /= Void
            line_exists: l /= Void
        ensure
            station_set: station = s
            line_set: line = l
    link (s: SIMPLE_STOP)
            — Сделать s следующей остановкой на линии.
        ensure
            right_set: right = s
            — Опущен инвариант: station /= Void and line /= Void; см. обсуждение
end
```

Запрос *station* дает связанную с остановкой станцию, а *line* – линию, к которой принадлежит остановка. Запрос *right* дает следующую остановку. С классом связаны две команды: *set_station_and_line* и *link*. Первая позволяет передать станции одновременно станцию и линию, вторая – следующую остановку на той же линии. Такие команды, главная цель которых – установить значения ассоциированных запросов (хотя они могут делать и большее), называются *получателями,* или *сеттерами* (*setters*).

Рассмотрим, как создается экземпляр этого класса. Предположим, что (наряду с *fancy_line*: *LINE*) мы уже объявили:

```
stop1: SIMPLE_STOP
```

Тогда в процедуре *build_a_line* мы можем создать остановку:

```
build_a_line
        — Построить воображаемую линию и подсветить её на карте
    do
        Paris.display
        — "Создать fancy_line"
        Paris.put_line (fancy_line)
        create stop1
        — "Создать следующие остановки и закончить построение fancy_line"
        fancy_line.highlight
    end
```

Оставшийся псевдокод детализируется двумя частями: сначала создается линия, а затем создаются и связываются остановки линии.

Оператор **create** *stop1* является **оператором создания**. Это базисная операция, создающая объекты во время выполнения программы. Ее эффект в точности соответствует смыслу слова **create**: создает объект и связывает его с сущностью, в данном случае *stop1*присоединяется

к новому объекту. На рисунках: все начинается состоянием, в котором *stop1*имеет значение void.

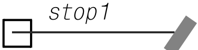


Рис. 6.12. Перед выполнением оператора создания

Выполнение присоединяет созданный для наших целей объект:

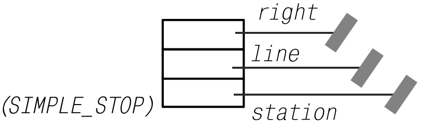


Рис. 6.13. После выполнением оператора создания

Оператор создания **create** не нуждается в указании типа создаваемого объекта, так как каждая сущность, такая как *stop1*, объявлена с указанием типа: *stop1*: *SIMPLE_STOP*. Тип создаваемого объекта определяется типом соответствующей сущности.

Как следствие предыдущего обсуждения – все ссылочные поля нового объекта получают значение **Void**. Мы можем связать их с фактическими объектами, используя команды-сеттеры класса *set_station_and_line* и *link*. Давайте построим все остановки *fancy_line*. Мы объявим три остановки:

```
stop1, stop2, stop3: SIMPLE_STOP
```

Заметьте, синтаксис позволяет объявлять одновременно несколько сущностей одного типа. Имена сущностей разделяются запятыми, а после двоеточия указывается их тип.

Числа на рисунке соответствуют порядку следования станций на нашей линии:

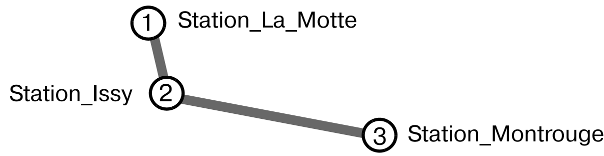


Рис. 6.14. Три станции на линии

Это позволяет нам написать следующую версию процедуры *build_a_line*:

```
build_a_line
        — Построить воображаемую линию и подсветить ее на карте.
    do
        Paris .display
        — "Создать fancy_line"
        Paris .put_line (fancy_line)
            — Создать остановки и связать каждую со своей станцией:
```

```
        create stop1
        stop1 .set_station_and_line (Station_Montrouge, fancy_line)
        create stop2
        stop2 .set_station_and_line (Station_Issy, fancy_line)
        create stop3
        stop3 .set_station_and_line (Station_Balard, fancy_line)
           - Связать каждую остановку со следующей за ней:
        stop1 .link (stop2)
        stop2 .link (stop3)
        fancy_line .highlight
    end
```

Заметьте, как последовательно сжимается псевдокод при добавлении новых операторов – реального кода, реализующего наши намерения. В конце весь псевдокод будет удален.

Два вызова *link* соединяют в цепочку первую остановку со второй, а вторую – с третьей. К третьей ничего не присоединяется, ее ссылка *right*, получившая при создании значение void, такой и останется. Это соответствует нашим желаниям: задавать таким способом последнюю остановку на линии.

Вызовы *set_station_and_line* должны удовлетворять предусловию метода, требующего, чтобы аргументы были присоединены к объектам:

- *Station_Montrouge* и другие станции, приходят от класса *TOURISM*, который позаботился о создании необходимых объектов;
- *fancy_line* будет присоединена, на что указывает оставшийся элемент псевдокода. Этот элемент будет детализирован ниже еще одним оператором создания **create**.

6.5. Процедуры создания

Процедура *build_a_line* использует простейшую форму создания:

```
create stop                                                          [2]
```

Этот оператор выполняет свою работу для сущности *stop* типа *SIMPLE_STOP*. Но можно его улучшить. Как показывает последняя версия процедуры, типичная схема создания остановки связывает ее с уже существующей станцией:

```
create stop                                                          [3]
stop.set_station_and_line (existing_station, existing_line)
```

Такой подход требует вызова метода непосредственно после вызова оператора создания, чтобы связать новый объект со станцией и линией. Объект, полученный в результате создания, как отмечалось, не имеет смысла, если он не связан со станцией и линией. Нам хотелось бы выразить это в виде инварианта:

```
invariant
        station_exists: station /= Void
        line_exists: line /= Void
```

При таком подходе класс становится некорректным, поскольку инвариант класса должен выполняться сразу же после создания, а этого не произойдет после простого выполнения **create** в **[2].**

Так появляются два повода слить два оператора в один – оператор создания и вызов *set_station_and_line*.

- Первый довод – удобство. Любой клиент, нуждающийся в создании остановки, обязан выполнить две команды. Если он забудет вторую, ограничившись только созданием объекта, то результатом будет некорректное поведение ПО и отказ при выполнении. Общее правило проектирования ПО заключается в том, что следует избегать создания элементов, требующих специальных предписаний: «*Когда вы сказали А, то не забудьте после этого сказать Б*» (программисты, как и обычные люди, не всегда читают инструкции и неукоснительно следуют им). Лучше обеспечить операцию, избавляющую от необходимости изучения хитроумного интерфейса.
- Второй довод – корректность. Мы хотим верить, что экземпляры класса непосредственно с момента создания являются согласованными, в данном случае имеют станцию и линию.

Для удовлетворения этих концепций мы можем объявлять в классе одну или несколько **процедур создания**. Процедурой создания является команда, которую клиент должен вызвать всякий раз, когда он создает экземпляр класса, убедившись при этом, что экземпляр класса должным образом инициализирован, удовлетворяя, в частности, всем инвариантам класса.

Введем процедуру создания, *set_station_and_line*, и объявим теперь наши остановки как

```
stop1, stop2, stop3: STOP
```

Вместо класса *SIMPLE_STOP* теперь используется класс *STOP*, в котором объявлена процедура создания, заменяющая прежнюю процедуру **create [2]**. Теперь вместо двух операторов, как в **[3]**, можно просто вызывать процедуру создания:

```
create stop1.set_station_and_line(Station_Montrouge, fancy_line)        [4]
```

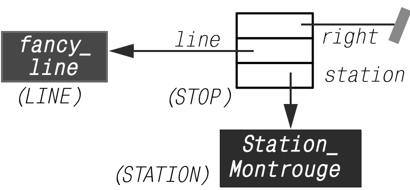


Рис. 6.15. После создания, использующего процедуру создания

Единственная разница между *STOP* и его предшественником: *STOP* имеет желаемый инвариант *station* /= **Void** и объявляет метод *set_station_and_line* как процедуру создания. Вот как выглядит теперь интерфейс класса, в котором изменилось немногое:

```
class STOP create
    set_station_and_line
feature
    station : STATION
            -- Станция, которую эта остановка представляет.
    line : LINE
            -- Линия, на которой находится эта остановка.
    right : STOP
            -- Следующая остановка на той же ассоциированной линии.
    set_station_and_line (s: STATION; l:LINE )
            -- Связать эту остановку с s и l.
        require
            station_exists: s/= Void
            line_exists: l/= Void
        ensure
            station_set: station =s
            line_set: line =l
    link(s : STOP)
            -- Сделать s следующей остановкой на той же линии
        ensure
            right_set : right =s
invariant
        station_exists: station /= Void
        line_exists: line /= Void
end
```

Та же процедура, что и прежде, но теперь в ранге процедуры создания

В верхних строчках интерфейса класса появилось новое предложение:

```
create
    set_station_and_line
```

Здесь использовано знакомое ключевое слово **create**, а далее перечисляется одна из команд класса, *set_station_and_line*. Это предложение говорит клиенту-программисту, что класс рассматривает *set_station_and_line* как процедуру создания. В предложении **create** указана единственная процедура создания, но разрешается вообще не задавать процедур создания или задать несколько таких процедур, поскольку могут существовать различные способы инициализации вновь созданного объекта.

Следствием включения такого предложения в интерфейс класса является то, что клиент не может теперь при создании объекта задействовать базисную форму **create** *stop* **[2].** Ему *необходимо* использовать одну из специальных процедур создания в форме **[4]**.

Это правило позволяет автору класса заставить клиента выполнить подходящую инициализацию всех создаваемых экземпляров класса. Оно тесно связано с инвариантами и требованием, чтобы непосредственно после создания объект удовлетворял всем условиям, налагаемыми инвариантами класса. В нашем примере инвариантами являются:

```
station_exists: station /= Void
line_exists: line /= Void
```

Предусловие *set_station_and_line*, в свою очередь, требует выполнения этих условий. Это общий принцип.

Почувствуй методологию: Принцип создания

> Если класс имеет нетривиальный инвариант, то класс должен иметь одну или несколько процедур создания, гарантирующих, что каждый создаваемый процедурой экземпляр удовлетворяет инварианту.

Инвариант, эквивалентный **True**, является тривиальным. К тривиальным относится и любое свойство, предполагающее инициализацию полей значениями по умолчанию (ноль для числовых переменных, **False** – для булевских, **Void** – для ссылок).

Даже в случае отсутствия сильных инвариантов может быть полезно обеспечить в классе процедуры создания, позволяющие клиентам комбинировать создание с инициализацией полей. Класс *POINT*, описывающий точки на плоскости, может предоставить клиентам процедуры создания *make_cartesian* и *make_polar*, каждая с двумя аргументами, обозначающими координаты точки. Клиент может задать точку, указав либо декартовы координаты, либо – полярные координаты точки.

В некоторых случаях, *POINT* является тому примером, можно допускать обе формы создания **[2]** и **[4].** Этот случай описывается следующим образом:

```
class POINT create
    default_create, make_cartesian, make_polar
feature
...
end
```

Здесь *default_create* – это имя метода (наследуемого всеми классами от общего предка) без аргументов, который по умолчанию помимо основной работы ничего больше не делает. Чтобы использовать эту процедуру, можно написать:

```
create your_point.default_create
```

Это можно выразить короче, в форме **[2]**:

```
create your_point
```

Эта форма является корректной наряду с другими формами:

```
create your_point.make_cartesian (x, y)
create your_point.make_polar (r, t)
```

Общее правило таково:

- если у класса нет предложения **create**, то это эквивалентно указанию одной из форм
  ```
  create
      default_create
  ```
  перечисляющей *default_create* в качестве единственной процедуры создания;
- соответственно, оператор создания может быть записан в краткой форме **[2] create** *x* без указания имени процедуры, либо в полной форме
  ```
  create x.default_create
  ```

Так что концептуально можно полагать, что каждый класс имеет процедуру создания, необходимую для создания его экземпляров.

Чтобы полностью построить *build_a_line*, нам осталось уточнить последнюю строку псевдокода: — “Create *fancy_line*”. Это соответствует вызову еще одного оператора создания

```
create fancy_line.make_metro ("FANCY")
```

Здесь используется *make_metro* – одна из процедур создания класса *LINE*, которая создает линию метро, принимая в качестве аргумента имя этой линии, принадлежащее строковому типу.

Так как все это доступно как часть предопределенных примеров, то хорошая идея – пойти и прочесть окончательный текст.

Время чтения программы!

Создание и инициализация линии
Рассмотрите текст *build_a_line* в классе *LINE_BUILDING* и убедитесь, что вы все прекрасно понимаете.

Как следствие предшествующего обсуждения, следует помнить, что необходимо создание объекта.

Создание экземпляра класса

- Если класс не имеет предложение **create**, используйте базисную форму, **create** *x* [2].
- Если класс имеет предложение **create**, перечисляющее одну или несколько процедур создания, используйте

```
create x.make (...)              –[4]
```

где *make* – это одна из процедур создания, а в скобках указываются подходящие аргументы для *make*, если они присутствуют. Необходимо указать правильное число аргументов и их правильные типы. Необходимо также, чтобы аргументы удовлетворяли предусловию для *make*, если оно указано.

6.6. Корректность оператора создания

В соответствии с принципами Проектирования по Контракту для каждого изучаемого оператора мы должны точно знать:

- как корректно использовать оператор: его *предусловие*;
- что мы получим в результате выполнения оператора: его *постусловие*.

Помимо этого, классы (и циклы, как вскоре увидим) имеют инварианты, которые описывают свойства, сопровождающие выполнение операций.

Совместно свойства контракта определяют **корректность** любого программного механизма.

Приведем соответствующее правило для механизма создания.

Почувствуй методологию:

Правило корректности оператора создания

Для корректности оператора создания *перед* его выполнением должно выполняться:

предусловие процедуры создания.

Следующие свойства должны иметь место *после* выполнения оператора создания, вызванного целью *x* типа *C*;

1) *x* /= **Void**;
2) постусловие процедуры создания, если оно задано;
3) инвариант класса *C*, примененный к объекту, который присоединен к *x*.

Если процедура создания не указана, то базисная форма **create** *x* тривиально удовлетворяет свойствам 1 и 3, поскольку не имеет ни предусловия, ни постусловия.

Предусловие процедуры создания (предложение 1) не требует, чтобы *x* имел значение void. Не является ошибкой последовательно создать два объекта с одной и той же целью *x*:

```
create x
        – После создания x не является void (смотри предложение 2)
create x
```

Это специфический пример полезен для понимания, хотя с содержательной точки зрения имеет изъян, поскольку объект, созданный первым оператором, будет немедленно забыт при выполнении второго оператора:

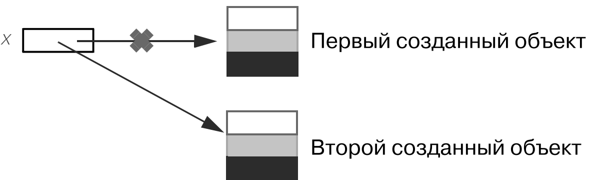


Рис. 6.16. Создание двух объектов

Второй оператор создания перенаправит ссылку на второй объект, так что первый объект станет бесполезным (вскоре мы поговорим подробнее о таких сиротливых объектах, не присоединенных ни к одной сущности).

Хотя два последовательных оператора создания в предыдущем примере не имеют смысла, варианты этой схемы могут быть полезными. Например, тогда, когда между операторами создания выполняются другие операторы, выполняющие полезные операции над первым объектом. Возможно, после создания первый объект будет куда-нибудь передан в качестве аргумента или записано его состояние.

Предложения со 2-го по 4-е определяют эффект выполнения оператора создания.

- Независимо от того, имела ли цель значение void, после вызова она это значение иметь не будет, поскольку результатом действия станет присоединение объекта (предложение 2).
- Если задана процедура создания, то ее постусловие будет иметь место для вновь созданного объекта (предложение 3).

- Помимо этого, объект будет удовлетворять инвариантам класса (предложение 4). Как установлено принципом инварианта, это требование является основой оператора создания: гарантируется, что объект, начинающий свою жизнь, удовлетворяет условиям согласованности, которые класс накладывает на все свои экземпляры.

Если инициализация по умолчанию не обеспечивает инвариант, то в обязанность процедуры создания входит корректировка ситуации таким образом, чтобы в итоге инвариант выполнялся.

6.7. Управление памятью и сборка мусора

В ситуации, изображенной на последнем рисунке, ссылка, которая была присоединена к объекту (первый созданный объект), затем была отсоединена и присоединена к другому. Что, хотелось бы знать, случилось с первым объектом? Хотя этот частный пример (два последовательных создания **create** *x* с одним и тем же *x*) не реалистичен, полезное ссылочное пересоединение является общеупотребительным и может поднимать те же самые вопросы. В свое время мы будем изучать *ссылочное присваивание*, такое как *x*:= *y*, чей эффект отображен на рисунке:

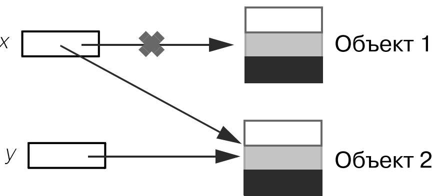


Рис. 6.17. Ссылочное переприсваивание

Оператор разрывает старую связь и присоединяет *x* к объекту, к которому присоединен *y*. Что случится с «Объектом 1»? Поставим более общий вопрос: мы рассмотрели способы *создания* объектов, а как объект может быть *удален* (*deleted*)?

Так как могут быть другие ссылки, присоединенные к «первому созданному объекту» или «Объекту 1», вопрос представляет реальный интерес. Когда ссылка на объект удаляется, как в наших примерах, то что случается с объектом, *если не остается никаких других ссылок на этот объект*? Здесь нет тривиального ответа, поскольку выяснение того, остались ли ссылки на данный объект, требует глубокого изучения всей программы в целом и понимания процесса выполнения программы. Возможны три подхода: *легкомысленный*, *ручной*, *автоматический*.

При легкомысленном подходе проблема просто игнорируется, неиспользуемые объекты остаются в памяти. Как следствие, это может привести к неконтролируемому росту занятой памяти. Такие *потери памяти* неприемлемы для постоянно работающих систем, используемых в различных встроенных устройствах, – потеря памяти на вашем мобильном телефоне может привести к остановке его работы. Все это справедливо для любой системы, создающей большое число объектов, которые постепенно становятся ненужными для системы. Легкомысленный подход неприемлем для любой нетривиальной системы.

Ручной способ управления снабжает программиста явными средствами возвращения объектов в распоряжение операционной системы. Программисты C++ могут, например, перед присваиванием *x*:= *y* освободить память, выполнив оператор *free* (*x*), сигнализирующий,

что объект не нужен программе, и операционная система может использовать по своему усмотрению память, занятую объектом.

Автоматический подход освобождает программиста, перекладывая эту работу на механизм, называемый «**сборщиком мусора**» (**garbage collector**), или коротко GC. На GC лежит ответственность возвращения недостижимых объектов. Сборка мусора выполняется как часть вашей программы. Более точно, она является частью системы периода выполнения (run time system) – множества механизмов, обеспечивающих выполнение программ.

Думайте о программе, как о параде, который идет по городу с музыкой, лошадьми и прочим, и о GC – как о бригаде уборщиков, следующей тем же маршрутом в нескольких сотнях метров позади и эффективно убирающей мусор, оставляемый парадом.

Реализация C++ обычно предпочитает ручной подход (из-за проблем, связанных с системой типов языка), но другие современные языки программирования обычно применяют автоматическую модель, используя сложно устроенные GC. Это верно для Eiffel, но справедливо и для Java и для Net-языков, таких как C#. Есть два главных довода для доминирования такого подхода.

- Удобство: включение в программу операций освобождения памяти значительно ее усложняет, поскольку необходимо выполнять интенсивные подсчеты, дабы убедиться, что данный объект стал бесполезным. При автоматическом подходе эти обязанности входят в задачу универсальной программы – GC, – доступной как часть реализации языка или надстройки над операционной системой.
- Корректность: поскольку подсчеты числа ссылок – вещь тонкая, ручной подход является источником весьма неприятных ошибок, когда операция free применяется к объекту, на который все еще остается ссылка в какой-либо далекой части программы. Как результат, программа может в какой-то момент работать некорректно с фатальными ошибками и прерыванием выполнения. Общецелевые GC эти проблемы в состоянии решить профессионально и эффективно не только для одной конкретной программы, но для всех программ.

Единственный серьезный аргумент против автоматической сборки мусора состоит в возможной потери эффективности. Освобождение объектов в любом случае требует определенного времени (исключая легкомысленный подход). Существует обеспокоенность, что GC будет прерывать выполнение в ответственный момент. Сегодня технология GC достигла такого уровня, что она не требует прерывания на полный цикл сборки – мусор собирается *понемногу* в некоторые моменты времени. Как результат, для большинства приложений прерывания становятся незаметными. Единственное остающееся беспокойство связано с системами «реального времени», встраиваемыми, например, в транспортные или военные устройства, где отклик системы требуется на уровне миллисекунд или менее и где недопустимы никакие посторонние задержки. Такие системы требуют особого подхода и не могут использовать многие другие преимущества современного окружения, включая динамическое создание объектов и виртуальную память.

В обычном окружении, где производится сборка мусора, ее доступность не означает, что нужно сорить, – программист может стать причиной неэффективного использования памяти. Приемы построения эффективных структур данных и алгоритмов, изучаемые в последующих главах, позволят избежать многих ловушек.

6.8. Выполнение системы

Заключительное следствие механизма создания состоит в том, что можно понять, как происходит процесс выполнения системы (программы в целом).

Все начинается со старта

Если понять, как создаются объекты, выполнение системы станет простой концепцией.

Определения: выполнение системы, корневой объект, корневой класс, корневая процедура создания

Выполнение системы состоит в создании экземпляра – **корневого объекта**, принадлежащего специальному классу системы, называемому **корневым классом**, – используя специальную процедуру создания этого класса, называемую **корневой процедурой создания**.

Достаточно, чтобы корневая процедура создания (будем называть ее просто **корневая процедура**) могла выполнять предписанные ей действия. Обычно она сама создает новые объекты и вызывает другие методы, которые могут в свою очередь делать то же самое, вызывая новые методы и создавая новые объекты, – так развивается процесс выполнения. Можно думать о нашей системе – множестве классов, как о множестве шаров на бильярдном столе. Корневая процедура наносит удар по первому шару, который бьет другие шары, которые, в свою очередь, бьют шары далее.

Рис. 6.18. Выполнение системы как игра в бильярд

Особенностью нашего бильярдного стола (нашей системы) является то, что могут создаваться новые шары, и в одном выполнении – одной игре – могут участвовать миллионы шаров, а не дюжина, как обычно.

Корневой класс, система и процесс проектирования

Корневой класс и корневая процедура запускают процесс, который основан на механизмах, лежащих в основе классов системы и их методов. Важно думать, что каждый класс интересен сам по себе, вне зависимости от любой частной системы и от выбора корневого класса и корневой процедуры. Как мы повторно увидим, классы являются машинами, каждая со своей ролью. Система станет некоторой сборкой таких машин, одну из которых мы выбрали для запуска выполнения.

Но классы существуют помимо системы. Класс может появляться в нескольких системах, составляя комбинации в каждом случае с разными классами.

Класс, методы которого представляют общий интерес для многих различных систем, называется **повторно используемым**. Классы, спроектированные для повторного использования, группируются в **библиотеки**. Но даже тогда, когда проектируется

> приложение со специфическими задачами, следует создавать классы, готовые к повторному использованию, насколько это возможно, поскольку всегда существует вероятность появления подобных задач.

В старых концепциях разработки ПО программа рассматривалась как монолитная конструкция, которая состоит из главной программы, разделенной на подпрограммы. При таком подходе трудно использовать старые элементы для новых целей, так как все они создавались как часть конструкции, предназначенной для одной цели. Требовались большие усилия для изменения программы, если менялась цель, как часто бывало на практике.

Более современные технологии архитектуры ПО основываются на ОО-идеях, которые и мы используем в этой книге, борясь с прежними пороками путем деления системы на классы (концепция более общая, чем подпрограмма), поощряя разработчиков уделять должное внимание каждому индивидуальному классу, делая его полным и полезным, насколько это возможно.

Для получения фактической системы, управляющей конкретным приложением, необходимо выбрать и скомбинировать несколько классов, затем разработать корневой класс и корневую процедуру, чтобы запустить процесс выполнения. Корневая процедура играет при этом роль главной программы. Разница методологическая: в отличие от главной программы корневой класс и корневая процедура не являются фундаментальными элементами проектирования системы. Это лишь частный способ запуска частного процесса выполнения, основанного на множестве классов, которые вы решили скомбинировать некоторым специальным образом. Множество классов остается в центре внимания.

Эти наблюдения отражают некоторые из ключевых концепций профессиональной инженерии ПО (в отличие от любительского программирования): **расширяемость** – простота, с которой можно адаптировать систему при изменении потребностей пользователя со временем; **повторное использование** – простота использования существующего ПО для нужд новых приложений.

Специфицирование корня

После короткого экскурса в принципы проектирования вернемся назад к более насущным проблемам. Один из непосредственных вопросов: а как специфицировать корневой класс и корневую процедуру системы?

Среда разработки – EiffelStudio – позволяет задать такие свойства системы. Это часть установок проекта (Project Settings), доступная в среде через меню File → Project Settings. Детали можно увидеть в приложении EiffelStudio.

Текущий объект и теория общей относительности

Видение выполнения системы позволяет нам понять фундаментальное свойство ОО-вычислений, которое можно было бы назвать общей относительностью, если бы этот термин не был уже использован много лет назад одним из выпускников ETH[1]. Главный вопрос: когда вы видите имя в классе, например, имя атрибута *station* в классе *SIMPLE_STOP*, то *что* оно означает в реальности?

В принципе, все, что нам известно, так это объявление и заголовочный комментарий:

```
station: STATION
        -- Станция, которую представляет эта остановка.
```

[1] На всякий случай уточню, что речь идет об Альберте Эйнштейне.

Не настораживает «эта остановка»? В операторе, использующем атрибут, таком как *station.set_name* («*Louvre*»), у какой станции мы изменяем имя?

Ответ может быть только относительным. Атрибут ссылается на текущий объект, появляющийся только во время выполнения. Неформально с этой концепцией мы уже встречались. Теперь пора дать точное определение.

Определение: Текущий объект

В любой момент выполнения системы всегда существует единственный текущий объект, определяемый следующим образом.

1. Корневой объект в момент начала выполнения является первым текущим объектом.
2. В момент начала квалифицированного вызова *x.f* (...), где *x* обозначает объект, этот объект становится новым текущим объектом.
3. Когда такой вызов завершается, предыдущий текущий объект вновь становится текущим.
4. Никакие другие операции не являются причиной изменения текущего объекта.
5. Для обозначения текущего объекта можно использовать зарезервированное слово **Current**.

Проследим за выполнением системы: корневой объект дается уже созданным; после возможного выполнения некоторых операций, в частности, создания новых объектов, может выполняться вызов, использующий в качестве цели один из созданных объектов. Он и становится на время текущим, в свою очередь, он может снова выполнить вызов на другой цели, которая и станет текущим объектом, и так далее. При завершении вызова предыдущий текущий объект восстанавливает свой статус текущего объекта.

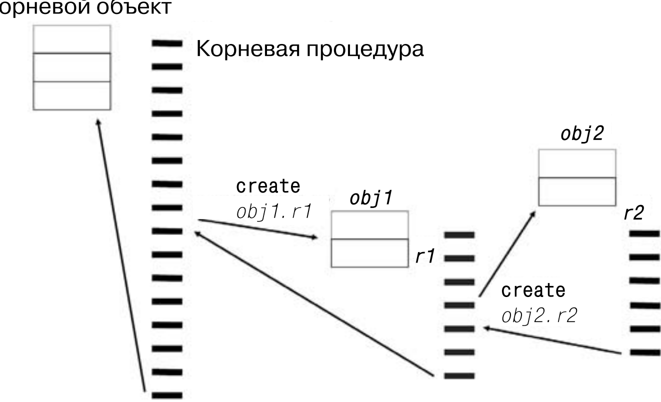


Рис. 6.19. Схема выполнения программы

Это отвечает на вопрос, что означает имя метода, когда оно появляется в операторах или выражениях (отличных от квалифицированных вызовов, когда оно появляется после точки, как *f* в *x.f* (...)): оно обозначает *метод, применимый к текущему объекту*.

В классе *SIMPLE_STOP* любое использование *station*, такое как *Console.show* (*station.name*), позволяющее отобразить имя станции остановки, означает поле «*station*» *текущего SIMPLE_STOP* объекта. Все это также объясняет слово «этот», используемое в заголовочных комментариях, вроде «Станция, которую *эта* остановка представляет».

Это соглашение является центральным для ОО-стиля программирования. Класс описывает свойства и поведение некоторой категории объектов. Он описывает эту цель путем описания свойств и поведения типичного представителя категории: текущего объекта.

Эти наблюдения приводят нас к обобщению понятия вызова. Рассмотрим оператор или выражение с точкой:

```
Console.show (station.name)       — Оператор
station.name                         — Выражение
```

В обоих случаях вызывается компонент, примененный подобно всем вызовам, к целевому объекту: объекту, обозначающему *Console* в первом примере, и *station* – во-втором. Но каков статус у самих объектов *Console* и *station*? Они также являются вызовами, целевой объект которых – текущий объект. Фактически, можно было бы писать:

```
Current.Console
Current.station
```

где, как выше было замечено, **Current** обозначает текущий объект. Нет необходимости, однако, использовать *квалифицированную* форму вызова в таких случаях; *неквалифицированная* форма *Console* и *station* имеет тот же самый смысл. Определения следующие:

Определения: квалифицированный и неквалифицированный вызов

Вызов метода называется **квалифицированным**, если цель вызова явно указана, используя нотацию с точкой, как в *x.f* (*args*).
Вызов называется **неквалифицированным**, если цель вызова не указана, таковой в этом случае является текущий объект, как в вызове *f* (*args*).

Важно осознавать, что многие выражения, чей статус до сих пор был для вас непонятен, фактически являются **вызовами** – неквалифицированными. Разнообразные примеры появлялись многократно:

```
Paris, Louvre, Line8                — В классе PREVIEW (глава 2)
south_end, north_end, i_th          — В инвариантах LINE (глава 4)
fancy_line                              — В данной главе
```

Все примеры относятся к этой категории. Инвариант класса *LINE* устанавливает:

```
south_end = i_th (1)
```

Это означает, что южный конец *текущей линии метро* является первой станцией на *той же линии*.

В вышеприведенном определении «текущего объекта» случай 4 говорит, что операции, отличные от квалифицированных вызовов, не изменяют текущий объект. Вот что истинно

для неквалифицированных вызовов: в то время как *x.f* (*args*) делает объект, присоединенный к *x*, новым текущим объектом на время вызова, неквалифицированный вызов в форме *f*(*args*) не является причиной смены текущего объекта. Но это вполне согласуется с тем, что неквалифицированный вызов можно записать в квалифицированной форме **Current**.*f* (*args*), где роль цели уже играет текущий объект.

Вездесущность вызовов: псевдонимы операций

Вызовы в наших программах фундаментальны и вездесущи. Наряду с квалифицированными вызовами в нотации с точкой, ясно говорящие о том, что имеет место вызов, простая нотация, подобная *Console* или *Paris*, также является примером вызовов – неквалифицированных.

Вызовы практически присутствуют даже в более обманчивых нарядах и масках. Возьмем, например, безобидно выглядящие арифметические выражения, подобные $a + b$. Вы можете предположить, что уж это выражение точно не является вызовом! Эти люди, помешанные на ОО-идеях, ничего не уважают, но всему есть предел, должно же быть что-то святое в программировании. К сожалению, предела нет. Нотация $a + b$ формально является специальной синтаксической формой, на программистском жаргоне – «*синтаксическим сахаром*» – записи **квалифицированного вызова** *a.plus* (*b*).

Соглашение просто. В классах, представляющих базисные числовые типы – возьмем для примера класс *INTEGER_32* из EiffelStudio, – вы сможете увидеть методы, такие как сложение, объявленные в следующем стиле:

```
plus alias "+"(other: INTEGER_32): INTEGER_32
        …Остальная часть объявления…
```

Спецификация **alias** обеспечивает необходимый синтаксический сахар, допуская форму $a + b$, известную как инфиксная нотация, как синоним *a.plus* (*b*) обычной ОО-нотации с точкой.

Нет смысла ограничиваться только целыми или другими базисными типами. Предложение **alias** можно добавлять в объявления:

- любого запроса с одним аргументом, такого как *plus*, допуская тем самым вызовы в инфиксной нотации, названной так потому, что знак операции разделяет операнды;
- любого запроса без аргументов, например, *unary_minus* **alias** «–», допуская вызов в префиксной нотации, скажем, $- a$, как синоним *a.unary_minus*.

Допустимо, чтобы один и тот же знак появлялся как для бинарных, так и для унарных операций. Так, знак «–» появляется в унарном и в бинарном запросе, так что можно писать $a - b$ вместо *a.minus* (*b*).

Допустимыми в **alias-**определениях могут быть знаки арифметических, булевских операций, операций отношения, но не только они; разрешается использовать последовательности символов, не включающие букв, цифр, знаков подчеркивания и не конфликтующие с предопределенными элементами языка. Этот механизм особенно полезен, когда создаются классы из проблемной области со своей символикой операций, что облегчает работу экспертов по данной проблематике с программным приложением.

В последующих главах мы увидим другие примеры того, как ОО-механизмы отступают от традиционной нотации, добавляя синтаксический сахар – нотацию с квадратными скобками, для записи доступа к элементам массива или словаря (хеш-таблицы): *your_matrix* [*i*, *j*] или *phone_book_entry* [«Jane»], представляющим аббревиатуры для вызова запросов – *your_matrix.item* (*i*, *j*), и *phone_book_entry.item* («Jane»). Для достижения подобного эффекта достаточно объявить в обоих примерах метод, названный *item*, как псевдоним: *item* **alias** «[]».

Объектно-ориентированное программирование является относительным программированием

«Общая относительность» природы ОО-программирования могла на первых порах сбивать с толку, так как не позволяла полностью понимать сами по себе программные элементы – их необходимо было интерпретировать в терминах охватывающего класса.

Я надеюсь, что теперь вы лучше понимаете картину в целом и ее влияние на написание отдельных классов. Мы отказываемся от монолитности – архитектуры «все в одной программе», в пользу децентрализованных систем, сделанных из компонентов, разрабатываемых автономно и комбинируемых многими различными способами.

6.9. Приложение: Избавление от void-вызовов

Это дополнительный материал, описывающий современное состояние исследований на момент публикации.

«Напасти», причиняемые void-вызовами, которые обсуждаются в этой главе, не являются неизбежными. Современная эволюция языков программирования, в особенности Spec C# (от Microsoft Research) и Eiffel, позволяют полагать, что эта проблема уходит в прошлое.

Версия стандарта ISO для языка Eiffel на самом деле является **void-безопасной**. Это *означает*, что *компилятор может гарантировать*, что во время любого выполнения системы не появятся void-вызовы. Поскольку реализация только появилась на момент написания, в этой книге не используется void-безопасная версия. Здесь представлены несколько элементарных понятий об этом безопасном варианте, позволяющие удовлетворить ваше любопытство.

В ISO-стандарте Eiffel тип, объявленный обычным образом, скажем, *CITY*, называется **присоединенным** типом, гарантирующим предотвращение void-ссылок. При объявлении *c: CITY* ссылка, обозначающая *c,* конструируется так, что в момент выполнения она всегда будет присоединенной. Тип только тогда допускает void-ссылки, если он объявлен как **отсоединяемый** тип с ключевым словом **detachable**, как в *s:* **detachable** *STOP*.

Оба примера репрезентативны, представляя образцы использования.

- Типы, представляющие объекты проблемной области, обычно должны быть присоединенными и, следовательно, исключают void: не бывает в жизни void-города.
- Типы, представляющие связные структуры данных, обычно должны поддерживать void-значения. Мы связывали в цепочку экземпляры *STOP*, создающие линию метро, заканчивая последнюю остановку void-ссылкой.

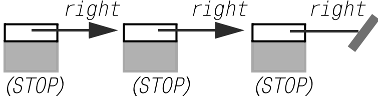


Рис. 6.20. Связанная линия

Гарантирование отсутствия void-вызовов основано на двух дополняющих приемах.

- Если сущность относится к присоединенному типу, она должна иметь ассоциированный механизм инициализации, запрещающий инициализацию по умолчанию значением **Void**, как было описано ранее в этой главе. Как следствие, перед первым вызовом *x.f* (...) сущность *x* всегда будет присоединена к объекту.
- Если *x* относится к отсоединяемому типу, любой вызов должен появляться в контексте, где гарантируется, что *x* отличен от void, например, **if** *x* /= **Void then** *x.f* (...) **end**. Су-

ществует фиксированное число обнаруживаемых безопасных возможностей, описанных в стандарте языка и известных как сертифицированные образцы присоединения (Certified Attachment Patterns или CAP).

Проектирование этого механизма поддерживает как совместимость, так и безопасность. Компилятор будет в большинстве случаев принимать обычный код – особенно старый код – таким, как он есть. Исключением является добавление **detachable**, когда тип должен поддерживать void-значения. Если компилятор отвергает такой код, то обычно это означает существование настоящей проблемы: код несет в себе риск void-вызова в момент выполнения. В таких ситуациях удаление причины отказа означает исправление ошибки, что облегчает работу программиста, а не создает ему помехи.

6.10. Ключевые концепции, изучаемые в этой главе

- Ссылка либо присоединена к объекту, либо имеет значение void.
- Вызов метода сущностью, такой как *x.f* (...), будет выполняться должным образом, только если *x* присоединен к объекту.
- Каждая ссылка изначально имеет значение void и остается таковой в отсутствие любых операций, подобных созданию, которые явно присоединяют объект.
- Void-ссылки служат для указания опущенной информации и для завершения связанных структур.
- Оператор создания, для которого указана цель *x,* создает новый объект и присоединяет *x* к этому объекту.
- Синтаксически оператор создания имеет вид **create** *x*, или – используя процедуру создания *p*, специфицированную в классе, – **create** *x.p* (*arguments*).
- Перед выполнением тела процедуры создания, если она задана, выполняется инициализация по умолчанию – поля вновь созданного объекта инициализируются стандартными значениями по умолчанию – ноль для чисел, void для ссылок.
- Оператор создания должен обеспечивать выполнение инварианта класса. Если инициализация по умолчанию не удовлетворяет этому требованию, то оператор должен использовать процедуру создания, исправляющую проблему.
- Выполнение системы состоит из создания экземпляра класса, специфицированного как «корневой класс». Для создания этого «корневого объекта» используется процедура создания класса, специфицированная как «корневая процедура».
- В любой момент выполнения существует *текущий объект*: объект, запустивший последний выполняемый метод.
- Вызов может быть квалифицированным, примененный к явно указанной цели, или неквалифицированный, примененный к текущему объекту.
- Каждое неквалифицированное упоминание метода должно пониматься (принцип «общей относительности» ОО-программирования) как применение к неявному объекту – текущему объекту, типичному представителю класса.
- Вызовы покрывают многие традиционные операции, некоторые из которых не используют нотацию вызова с точкой. В частности, операции в выражениях являются специальными случаями вызовов, записываемыми в инфиксной или префиксной нотации, что достигается введением псевдонимов (alias) для методов.
- Новые «void-безопасные» механизмы создают возможность (благодаря статическим проверкам, выполняемым компилятором) гарантировать отсутствие void-вызовов во время выполнения.

Новый словарь

Alias	Псевдоним	**Attached**	Присоединенный
Creation procedure	Процедура создания	**Current object**	Текущий объект
Detachable	Отсоединяемый	**Entity**	Сущность
Exception	Исключение	**Main program**	Главная программа
Failure	Отказ	**Library**	Библиотека
Extendibility	Расширяемость	**Operator alias**	Знак операции (псевдоним)
Pseudocode	Псевдокод		
Qualified call	Квалифицированный вызов	**Reference**	Ссылка
		Reusable	Повторно используемый
Reusability	Способность повторного использования	**Root class**	Корневой класс
		Root creation procedure	Корневая процедура создания
(= Root procedure)	Корневая процедура	**Root_object**	Корневой объект
Unqualified call	Неквалифицированный вызов	**Void reference**	Void-ссылка
		Void-safe	Void-безопасный

6-У. Упражнения

6-У.1. Словарь

Дайте точные определения всем терминам словаря.

6-У.2. Карта концепций

Добавьте новые термины в карту концепций, спроектированную в предыдущих главах.

6-У.3. Инварианты для точек

Рассмотрите класс *POINT* с запросами *x* и *y*, представляющими декартовы координаты, и *ro* и *theta*, представляющими полярные координаты. Напишите часть инварианта, включающую эти запросы. Вы можете использовать подходящие математические функции (функции тригонометрии) и полагать, что арифметические действия выполняются точно.

6-У.4. Current и Void

Может ли **Current** иметь значение **Void**?

6-У.5. Присоединенный и отсоединяемый

Приложение к этой главе описывает новый void-безопасный механизм, основанный на определении каждого типа либо как присоединенного, либо как отсоединяемого. Проанализируйте классы этой главы (и глав со 2-й по 4-ю) и установите, должны ли они всегда относиться к присоединенному или отсоединяемому типу, или решение определяется контекстом.

7

Структуры управления

К настоящему моменту у нас должно быть сформировано первое понимание того, как устроены *структуры данных* во время выполнения программы: это совокупность объектов, связанных ссылками. Пришло время посмотреть на *управляющие структуры*, определяющие порядок, в котором будут применяться операторы к этим объектам.

7.1 Структуры для решения задач

Возможно, вы знакомы с известной пародией на то, как учат инженеров.

Как вскипятить чайник воды

1. Если вода холодная: поставьте чайник на огонь, дождитесь, пока она закипит.
2. Если вода горячая: дождитесь, пока она остынет. Условие случая 1 выполнено. Примените случай 1.

Как способ кипячения воды эта техника не особенно эффективна, но дан прекрасный пример комбинирования некоторых из фундаментальных управляющих структур.

- *Выбор*: «Если данное условие выполняется, то делай одно, иначе – делай другое».
- *Последовательность*: «Делай вначале одно, потом другое».
- *Подпрограмма* позволяет нам именовать способ решения некоторой проблемы (возможно параметризованный) и повторно использовать его в подходящем контексте.

Полезно вспомнить обсуждение контрактов в предыдущих главах. Здесь мы отмечаем, что переход к случаю 1 из случая 2 возможен – что явно подчеркнуто при описании случая 2, – поскольку первый шаг случая 2 гарантирует выполнение **предусловия** случая 1 (вода холодная). Предусловия и другие контракты будут играть большую роль в получении правильных структур управления.

В этом примере в непринужденной манере устанавливается подходящий контекст изучения управляющих структур: они задают **приемы решения задачи**. Программа дает решение задачи; каждый вид управляющей структуры отражает частичную стратегию поиска решения задачи.

Задача всегда будет ставиться так: начиная с известного свойства K, требуется достичь некоторой цели G. В примере: свойство K – чайник с водой, цель G – вскипятить воду. Стратегии, обеспечиваемые управляющими структурами, состоят в редуцировании (приведении) задачи к нескольким более простым задачам. Вот пример.

- Можно применять управляющую структуру-**последовательность**, если удается сформулировать частную цель *I*, такую что обе новые проблемы проще исходной (достижение цели *G* непосредственно из *K*): достичь *I* из *K*, достичь *G* из *I*. Зная, как решить первую и вторую проблему, управляющая структура-последовательность решит вначале первую проблему, а затем вторую.
- Управляющая структура **выбора** представляет стратегию разбиения множества начальных возможных ситуаций *K* на две или более непересекающиеся области, так что становится проще решать задачу независимо на каждой из областей.
- Структура-**цикл**, которую мы еще увидим в примерах, является стратегией повторяющегося решения задачи на подмножествах (возможно, тривиальных). Подмножества расширяются до тех пор, пока не покроют всю область.
- Управляющая структура-**подпрограмма** (процедура или функция) является стратегией решения задачи, основанной на обнаружении того, что наша задача сводится к решению другой (часто более общей задачи), решение которой уже известно.

  Техника *рекурсии*, достаточно важная, чтобы посвятить ей отдельную главу, может быть применена, если можно сконструировать решение задачи из одного или нескольких решений той же самой задачи, но примененной к данным меньшего размера.

Так как программирование предназначено для решения задач, полезно изучать эти и другие структуры с этих позиций.

Для каждой из управляющих структур будем последовательно анализировать:

- общую идею, приводя *примеры*;
- *синтаксис* конструкции в соответствующем языке;
- *семантику*: эффект исполнения – как управляющая структура задает порядок выполнения входящих в нее операторов;
- *корректность*: правила, основанные на принципах Проектирования по Контракту. Это позволит нам убедиться, что семантика соответствует намерениям и что управляющая структура при выполнении дает имеющий смысл результат, не приводя к поломке выполнения программы или к другим неприятным следствиям.

7.2. Понятие алгоритма

Управляющие структуры определяют порядок операций в процессах, выполняемых компьютером. Такие процессы называются *алгоритмами*. Это одна из фундаментальных концепций информатики. Вам уже наверняка приходилось встречаться с этим термином, поскольку даже популярная пресса обсуждает такие темы, как «*криптографические алгоритмы*». Для изучения управляющих структур нам понадобится более точное определение.

Пример

В общих терминах алгоритм – это *описание* процесса вычислений, достаточное, чтобы его могла выполнить машина (для нас в роли машины выступает компьютер), осуществление процесса на любых входных данных без дальнейших инструкций.

Нам известно много различных алгоритмов. Сложите два числа:

$$\begin{array}{r} 687 \\ +\ 42 \\ \hline =729 \end{array}$$

Для решения этой задачи вы примените правила, скорее всего, не думая явно об этих правилах.

Почувствуй Арифметику

Сложение двух целых чисел

Процесс состоит из нескольких *шагов*, каждый шаг выполняется над определенной **позицией** чисел. Позиция для первого шага задается самой правой цифрой обоих чисел. Для каждого следующего шага его позиция – непосредственно слева от предыдущей позиции.

У каждого шага есть **перенос**. Начальный перенос равен нулю.

На каждом шаге пусть *m* – это цифра первого числа, стоящая в позиции данного шага, а *n* – это соответствующая цифра второго числа в той же позиции. Если какое-либо число не имеет цифр в данной позиции, то значение (*m* или *n*) равно нулю.

На каждом шаге процесс выполняет следующее:

1) вычисляет *s* – сумму трех значений: *m*, *n* и переноса;
2) если s меньше 10, запишет s в позицию шага для результата и установит значение переноса равным нулю;
3) если s равно 10 или больше, запишет s – 10 в позицию шага для результата и установит значение переноса равным 1. Записанное значение s – 10 является цифрой, так как s не может быть больше 19.

Процесс заканчивается, когда в позиции шага нет цифр у обоих чисел и перенос равен нулю.

Этап 3 основан на предположении: «*s* не может быть больше 19». Без него процесс не имел бы смысла, так как мы хотим писать в результат цифру за цифрой. Чтобы гарантировать корректность алгоритма, нам нужно доказать, что это свойство выполняется на каждом шаге процесса. Действительно, *m* и *n* являются цифрами, не могут быть больше 9, а их сумма не больше 18. Так как перенос не больше 1, *s* не больше 19. Это пример **инвариантного свойства**, концепции, которую будем изучать детально при рассмотрении циклов.

Явно и точно: алгоритмы против рецептов

Будучи менее точной, чем принятый стандарт публикации алгоритмов, предыдущая спецификация все же более пунктуальна, чем большинство предписаний, которые нам приходится использовать с различной степенью успеха в обычной жизни. Вот пример инструкции на трех языках на пакете куриного супа с овощами.

На немецком и французском языках инструкция говорит: «*Залейте овощи одним литром холодной воды, добавьте две столовые ложки масла и соль*». Рецепт не точен, поскольку «столовые ложки» бывают разными, и сколько нужно соли, тоже не указано. Что более удивительно, так это отсутствие ключевой инструкции: если вы хотите получить результат, то без огня не обойтись. Только итальянская версия упоминает эту деталь – «*Готовьте в соответствии с указанным временем*», что придает смысл рисунку.

Такие инструкции ориентированы на человека, его интерпретацию; отсутствие явно заданных шагов не представляет проблемы, поскольку большинству пользователей ясно, что еду не приготовишь без кипячения и что картинка задает время приготовления супа (даже я догадался). Но то, что годится для кухонных рецептов, недостаточно для алгоритмов. Вы

PREPARAZIONE E TEMPI DI COTTURA
ZUBEREITUNG - PREPARATION

Versate le verdure ancora surgelate in 1 litro abbondante d'acqua fredda con 2 cucchiai d'olio, salate e cuocete secondo i tempi indicati.

Tiefgefrorene Gemüse in einen Liter kaltes Wasser geben, 2 Esslöffel Öl und Salz hinzufügen.

Verser les légumes surgelés dans 1 litre d'eau froide, ajouter deux cuillers à soupe d'huile et du sel.

Рис. 7.1. Это не алгоритм (перевод приведен в тексте).

должны специфицировать каждую операцию, каждую деталь процесса, и их нужно специфицировать в форме, не оставляющей двусмысленностей, никакой свободы для предположений.

Свойства алгоритма

Для алгоритмов в противоположность неформальным рецептам справедливы свойства, задаваемые следующим определением.

Определение: Алгоритм

Алгоритм является спецификацией процесса, действующей на (возможно пустом) множестве данных и удовлетворяющей следующим пяти правилам.

A1. Спецификация определяет применимое множество данных.

A2. Спецификация определяет множество элементарных действий, из которых строятся все шаги процесса.

A3. Спецификация определяет возможный порядок или порядки, в которых процесс может выполнять свои действия.

A4. Спецификация элементарных действий (правило A2) и предписанный порядок (правило A3) основаны на точно определенных соглашениях, позволяя процессу выполняться автоматически, без вмешательства человека. Гарантируется, что на одном и том же множестве данных результат будет одинаковым для двух автоматов, следующих тем же соглашениям.

A5. Для любого множества данных, для которых процесс применим (по правилу A1), гарантируется завершение процесса после выполнения конечного числа шагов алгоритма.

Вышеприведенный метод сложения целых обладает требуемыми свойствами:

A1: описывает процесс, применимый к некоторым данным, и специфицирует вид этих данных: два целых числа, записанные в десятичной нотации;

A2: процесс основан на хорошо определенных базисных действиях: установить значение, равное нулю или известному числу, сложить три числа (цифры), сравнить число с 10;
A3: описание задает порядок выполнения базисных действий;
A4: правило является точным. Этой точности должно быть достаточно для любых двух людей, понимающих и применяющих алгоритм одинаковым способом. Хотя, как отмечалось, оно может быть недостаточно точным для других целей;
A5: для любых применимых данных – два числа в десятичной нотации – процесс завершится после конечного числа шагов. Это интуитивно ясно, но должно быть тщательно проверено. Мы увидим, как это делается, показав, что выражение $M - step + 1$ является *вариантом*.

В определении правила A3 упоминается «порядок или *порядки*» шагов. *Последовательный* и *детерминированный* алгоритм определяет единственный порядок шагов для любого возможного выполнения. Но это не единственная возможность.

- *Недетерминированные* алгоритмы для некоторых шагов определяют множество действий, одно из которых должно быть выполнено, но алгоритм не определяет, какое именно. *Вероятностный* алгоритм определяет как специальный случай случайную стратегию для такого выбора.
- *Параллельные* алгоритмы задают для некоторых шагов множество действий, выполняемых параллельно, что применимо для вычислительных сетей и многоядерных компьютеров.

В этой книге будут рассматриваться только последовательные детерминированные алгоритмы.

Алгоритмы против программ

В свете приведенного определения алгоритма хотелось бы понять, что же отличает алгоритмы от программ? Базисные концепции одни и те же.

Иногда говорят, что разница лежит в уровне абстракции: программа предполагает выполнение на специальной машине, в то время как алгоритм задает абстрактное определение процесса вычислений независимо от любого вычислительного устройства. Это имело некоторый смысл несколько десятилетий назад, когда программы записывались в кодах конкретного компьютера. Алгоритмы тогда служили для выражения сущности программ: вычислительный процесс описывался независимо от любого компьютера. Но та точка зрения не применима сегодня.

- Для представления программ мы можем использовать ясную, высокоуровневую нотацию, определенную на уровне абстракции, который существенно превосходит детали любого компьютера. Нотация языка Eiffel, используемая в этой книге, является примером.
- Для представления алгоритма способом, полностью отвечающим требованиям определения, в частности, требованию точности – условие A4 – также необходима нотация с точно определенным синтаксисом и семантикой, что в конечном итоге делает эту нотацию эквивалентной языку программирования.

Практическое описание алгоритмов часто опускает специфицирование некоторых деталей, таких как выбор структур данных, которые программа не может опустить, так как без этого невозможна компиляция и выполнение. Эта практика, кажется, обосновывает довод о более высоком уровне абстракции алгоритмов в сравнении с программами. Но для алгоритмов это только полезное соглашение, облегчающее их публикацию. В описаниях, предназначенных для публикации, снижаются требования к точности (условие A4), но каждый читатель понимает, что для получения алгоритма в настоящем смысле необходимо восстановить опущенные детали.

Так что мы не можем полагать, что уровень абстракции отличает алгоритмы от программ. Более важны другие два отличия – или нюанса.

- Алгоритм описывает единственный вычислительный процесс. Десятилетия назад в этом же состояла цель каждой типичной программы: «*Начислить зарплату*!». Сегодня программа включает множество алгоритмов. Мы уже видели образец в системе Traffic (отобразить линию, анимировать линию, отобразить маршрут ...), и таких примеров множество. Подобное наблюдение применимо к любому серьезному программному продукту. Вот почему в этой книге чаще используется термин «*система*», а не «программа» (которую часто по-прежнему воспринимают как решение одной задачи).
- Как важно в программе описать шаги процесса обработки, так же важно и описать в ней структуры данных – в ОО-подходе **структуру объектов**, – к которым эти шаги применяются. Этот критерий не является абсолютным, так как реально нельзя отделить алгоритмические шаги от структур данных, которыми они манипулируют. Но при описании программистских концепций иногда хочется большее внимание уделить аспекту обработки – алгоритму в узком смысле этого термина, – а иногда аспекту данных. Это объясняет заглавие классической книги по программированию Никласа Вирта (опубликованной в 1976):
  Algorithms + *Data Structures* = **Programs**.
  На русском языке: «Алгоритмы + Структуры данных = Программы»

Рис. 7.2. Вирт (2005)

ОО-подход к конструированию ПО отводит центральную роль данным, более точно – типам объектов: классам. Каждый алгоритм присоединен к некоторому классу. Eiffel применяет это правило без исключений: каждый алгоритм, который вы пишете, появляется как *метод* (*feature*) некоторого класса. Этот подход оправдан по соображениям качества создаваемого ПО, которое будем анализировать в последующих главах. Как следствие, в этой книге мы будем изучать алгоритмы и аспекты данных в тесном взаимодействии.

Структуры управления, рассматриваемые в этой главе, являются одним из примеров алгоритмических концепций, не связанных напрямую с конкретным видом структуры данных.

7.3. Основы структур управления

Спецификация алгоритма должна включать элементы двух типов:

- элементарные шаги выполнения (требование A2 определения алгоритма);
- порядок их выполнения (требование A3).

Управляющие структуры отвечают второй из этих потребностей. Более точно:

Определения: Поток управления, Управляющая структура

Расписание, задающее порядок следования операций во время выполнения, называется **потоком управления.**
Структура управления или управляющая структура – это программная конструкция, воздействующая на поток управления.

Как уже отмечалось, существуют три фундаментальные формы структур управления.

- **Последовательность**, которая состоит из операторов, перечисленных в определенном порядке. Ее выполнение состоит в выполнении каждого из этих операторов в том же порядке.

  Эту структуру неявно мы уже многократно использовали в большинстве примеров, так как писали операторы в предположении, что они выполняются в заданном порядке.

- **Цикл**, который содержит последовательность операторов, выполняемую многократно.
- **Выбор,** состоящий из условия и двух последовательностей операторов. Выполнение состоит из выполнения либо одной, либо другой последовательности, в зависимости от того, принимает ли условие – булевское выражение – значение **True** или **False**. Структура может быть обобщена на случай выбора из нескольких возможностей.

Механизмы работы операторов программы используют три фундаментальные возможности компьютеров (последовательность, цикл, выбор).

- Выполнение *всего* множества предписанных действий в заданном порядке.
- Выполнение *единственного* предписанного действия, или как вариант – многократное выполнение этого действия.
- Выполнение *одного* из множества предписанных действий в зависимости от заданного условия.

Такие структуры управления предполагают, что программа в каждый момент времени выполняет только одно действие. Для компьютеров с несколькими процессорами или для компьютера, выполняющего несколько программ одновременно, можно говорить о *параллельном* выполнении, приводящем к новым структурам управления, которые здесь мы изучать не будем.

Наши базисные структуры можно комбинировать без всяких ограничений, так что можно задать структуру выбора, каждая ветвь которой включает цикл или структуру выбора, а они, в свою очередь, включают другие структуры. Процесс вычислений состоит из операторов, которые сгруппированы в структуры управления, описывающие план работы периода выполнения, – описание этого процесса и составляет алгоритм.

Эти понятия являются предметом следующих разделов. В дополнение рассмотрим еще две формы управляющих структур.

- *Оператор перехода*, также известный как оператор **goto**, потерявший благосклонность у программистов – мы увидим почему, – но все еще играющий роль в системе команд компьютера.
- *Обработка исключений*, обеспечивающая способы восстановления после возникновения событий, прерывающих нормальный поток управления, таких как например, void-вызовы.

После того, как алгоритм определен, возникает естественное желание превратить его в программу, дать ей имя и использовать ее, зная это имя. Такое группирование известно как **создание подпрограммы** – фундаментальной формы структурирования, достижимой с пози-

ций управления. Подпрограммы в результате позволяют нам создавать новые структуры управления, создавая как новую абстракцию некоторую комбинацию существующих структур. Они являются предметом следующей главы.

7.4. Последовательность (составной оператор)

Структура «последовательность», применимая к образцам решения задач, известна каждому: задав одну или несколько промежуточных целей, можно постепенно, шаг за шагом двигаться по направлению к цели. Если ставится только одна промежуточная цель, то мы решаем две раздельные задачи:

- достичь промежуточной цели, начиная с исходных предположений;
- достичь финальной цели, начиная с промежуточного рубежа.

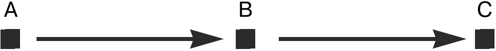


Рис. 7.3. Достижение цели, проходя промежуточный рубеж

Для более общего случая, с n промежуточными целями, будем выполнять $n + 1$ шагов, где на шаге i (для $2 \leqslant i \leqslant n$) достигается i-я промежуточная цель, полагая, что предыдущие цели достигнуты.

Примеры

В рассматриваемой нами проблемной области путешествия по городу типичным примером последовательности является возможная стратегия перемещения из точки *a* в точку *b*.

1. Найти на карте станцию метро *ma*, ближайшую к *a*.
2. Найти на карте станцию метро *mb*, ближайшую к *b*.
3. Пройти от *a* к *ma*.
4. Проехать на метро от *ma* до *mb*.
5. Пройти от *mb* к *b*.

Эта стратегия предназначена для человека, не для программы. Программа также может построить маршрут, используя введенные понятия. Маршрут (*route*), как вы помните, состоит из этапов (*leg*). Зададим, соответственно, следующие объявления:

```
full: ROUTE
walking_1, walking_2, metro_1: LEG
```

Маршрут можно построить, используя следующую последовательность операторов:

```
                                                    — Версия 1
create walking_1.make_walk (a, ma)
create walking_2.make_walk (mb, b)
create metro_1.make_metro (ma, mb)
create full.make_empty
full.extend (walking_1)
full.extend (metro_1)
full.extend (walking_2)
```

Здесь мы в полной мере используем те преимущества, которые дает нам возможность иметь различные процедуры создания в классе *LEG*:

- *make_walk*, создает пеший этап от одного места к другому;
- *make_metro*, создает этап поездки на метро от одной станции до другой (с предусловием, требующим существования линии, которая проходит через обе станции, так как один этап метро должен быть полностью на одной линии).

Мы используем также процедуру создания и метод класса *ROUTE*:

- процедуру создания *make_empty*, строящую пустой маршрут;
- команду *extend*, добавляющую этап в конец маршрута.

Время программирования

Создание и анимация маршрута

Используя вышеприведенную схему, напишите и выполните программу, создающую маршрут от Елисейского дворца (*Elysee_palace*) до Эйфелевой башни (*Eiffel_tower*). Имена обоих мест определены как компоненты класса *TOURISM*. Анимируйте полученный маршрут.

Вставьте соответствующие программные элементы и последующие в этой главе в новый класс, названный *ROUTES*. Имя системы для примеров и экспериментов с ней – *control*.

Начиная с этого и для всех последующих упражнений, больше не будут даваться пошаговые инструкции, как писать, компилировать и выполнять примеры, если только не понадобятся новые механизмы EiffelStudio, о которых ранее не шла речь. При возникновении вопросов обращайтесь к приложению.

Составной оператор: синтаксис

Управляющая структура «последовательность» не является новинкой этой главы: мы уже встречались с ней многократно – фактически, начиная с самого первого примера, – только не используя это имя. Мы просто писали несколько операторов в порядке предполагаемого выполнения, как в примере:

```
Paris.display
Louvre.spotlight
Metro.highlight
Route1.animate
```

Так как часто полезно рассматривать *последовательность* операторов как *один оператор*, например, для того чтобы сделать ее частью другой управляющей структуры, – последовательность часто называют **составным оператором**.

Правило синтаксиса очень простое.

Синтаксис

Составной оператор

Для задания последовательности – *составного оператора*, содержащего ноль или более операторов, – пишите их последовательно друг за другом, в желаемом порядке выполнения, разделяя при необходимости символами «точка с запятой».

До сих пор мы не пользовались символами «точка с запятой». Правила стиля говорят, что фактически о них особенно беспокоиться не нужно.

Почувствуй стиль

Точки с запятой между операторами

- **Опускайте** точки с запятой, если (так следует поступать в большинстве случаев) каждый оператор появляется на отдельных строчках.
- В редких случаях появления нескольких операторов на одной строке (операторы короткие, и у вас есть причины экономии числа строк) **всегда** разделяйте операторы символом «точка с запятой».

Так что, если вам нужно напечатать выше приведенную «Версию1» и у вас туго с бумагой (или ваш босс – рьяный защитник окружающей среды), то можно написать последние три оператора так:

```
full.extend (walking_1); full.extend (metro_1); full.extend (walking_2)
```

Чтобы так поступать, должны быть веские доводы. На самом деле, обычно один оператор располагается на строке, так что можно забыть о точках с запятой.

Важно помнить, что разделение на строчки **не несет никакой семантики**; конец строки – это просто символ, с тем же эффектом, как пробел или табуляция. Так что ничто не мешает вам написать и так:

```
full.extend (walking_1) full.extend (metro_1) full.extend (walking_2)
```

Безобразно!

Ничто, за исключением хорошего вкуса, элементарного здравого смысла и официальных правил стиля, принятых в организации. Не стоит сбрасывать со счетов и тех, кто будет читать ваш программный текст позже, в особенности двух читателей: преподавателя (если программа пишется в рамках учебного курса) и вас самих после нескольких дней, недель или месяцев.

Даже записывая операторы на отдельных строчках, некоторые нервничают из-за отсутствия разделителей, возможно, по той причине, что многие языки программирования строго требуют их присутствия в одних места, и запрещают их появление – в других. Проведите простой тест: напишите две одинаковые программы по одному оператору на строке, но в первой завершайте каждую строку точкой с запятой, а во второй не используйте эти символы. Поставьте эти программы рядом и сравните – вы убедитесь, что текст второй программы выглядит чище и лучше читается.

Если вы используете точки с запятой и ошибочно поставите лишнюю, то вреда это не причинит, поскольку *оператор_1; ; оператор_2* формально воспринимается как три оператора, в котором второй является *пустым* оператором, имеющим семантику «ничего не делать», так что такая конструкция не создаст трудностей. Конечно, лучше чистить текст и удалять все ненужные элементы.

Составной оператор: семантика

Поведение в период выполнения следует из самого смысла имени конструкции.

Семантика

> **Составной оператор**
> Выполнение последовательности операторов состоит из выполнения каждого оператора в заданном порядке.

Заметьте, что синтаксис говорит, что последовательность может быть пустой без операторов – тогда она эквивалентна пустому оператору и ничего не делает. Это не особенно впечатляет, но иногда бывает полезным при построении сложных структур управления.

Избыточная спецификация

Возможно, вы заметили, что в вышеприведенном примере («Версия 1») выбранный порядок является одним из возможных. Например, можно добавлять в маршрут каждый этап непосредственно после его создания:

```
                                                    —Версия 2
    — Создать маршрут:
create full .make_empty
    — Создать и добавить первый этап:
create walking_1 .make_walk (a, ma)
full .extend (walking_1)
    — Создать и добавить второй этап:
create metro_1 .make_metro (ma, mb)
full .extend (metro_1)]
    — Создать и добавить третий этап:
create walking_2 .make_walk (mb, b)
full .extend (walking_2)
```

Возможны и другие порядки. Единственное ограничение: любой оператор, использующий объект, должен появляться после создания этого объекта; и мы добавляем этапы в правильном порядке.

Использование «последовательности» часто задает **излишнюю спецификацию**, поскольку приводимое решение не является наиболее общим. Это не причиняет вреда системе, но следует сознавать, что решение является одним из множества возможных.

Когда скорость выполнения является значимой, иногда можно ускорить работу, выполняя группу операторов параллельно. Четыре начальных оператора создания могут выполняться параллельно с тем же эффектом. Параллельность, однако, дело тонкое, программисты обычно в таких простых случаях распараллеливанием не занимаются, но хорошие компиляторы могут создавать параллельный код, если аппаратура поддерживает такую возможность, что теперь становится нормой для современных многоядерных процессоров.

Составной оператор: корректность

Мы знаем, что метод может иметь контракт, включающий предусловие и постусловие. Эти свойства управляют вызовом метода, такого как, например, *full.extend* (*walking_1*). Предусловие говорит клиенту (вызывающему метод), что он должен гарантировать, чтобы его корректно обслужили. Постусловие говорит клиенту, на что он может полагаться по завершении корректного вызова.

Аналогично каждая управляющая структура, рассматриваемая в этой главе, обладает связанным с ней правилом корректности, которое, используя контракты (предусловия и постусловия операторов, входящих в структуру), определяет результирующий контракт для структуры в целом. Для составного оператора правило корректности отражает свойство поочередного выполнения составляющих операторов.

Корректность

Составной оператор

Для корректности составного оператора:

- необходимо выполнение предусловия первого оператора, если таковое задано, до начала выполнения составного оператора;
- из истинности постусловия каждого оператора должна следовать истинность предусловия следующего оператора, если таковое имеется;
- из истинности постусловия последнего оператора должна следовать истинность желаемого постусловия составного оператора в целом.

Специальный случай: пустой составной оператор всегда корректен, но не добавляет никаких новых постусловий.

В нашем примере можно проверить контракт метода *extend* класса ROUTE, получив его из EiffelStudio. С некоторыми опущенными постусловиями он выглядит так:

```
extend (l: LEG)
    require
        leg_exists: l /= Void
    ensure
        lengths_added: count = old count + 1
```

Каждая процедура создания в форме **create** *x* или **create** *x.make* (...) гарантирует истинность условия *x /= Void* после ее завершения. Таким образом, предусловие корректности для составного оператора выполняется как в «Версии 1», так и в «Версии 2», но может не иметь места, если изменить порядок выполнения операторов:

```
                                                    — Версия 3
    — Создать маршрут:
create full .make_empty
    — Создать и добавить первый этап:
full .extend (walking_1)
create walking_1 .make_walk (a, ma)
```

В том месте, где используется *walking_1*, соответствующий *LEG* объект еще не существует, так что попытка добавить его к маршруту будет ошибочной.

7.5. Циклы

Наша вторая структура управления – цикл – отражает одно из самых замечательных свойств компьютеров: их способность повторять операцию или варианты этой операции много раз, по человеческим меркам – *очень* много раз.

Типичным примером цикла является схема анимации, подсвечивающая линию метро и отображающая красную точку на каждой станции поочередно через каждые полсекунды. Система *Show_line* в поставке Traffic реализует этот сценарий. Вы можете выполнить ее, если желаете. Эффект выполнения одного из промежуточных шагов показан на рисунке:

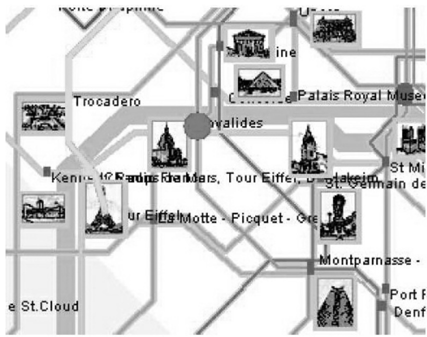


Рис. 7.4. Подсветка станции

Этот эффект достигается благодаря циклу. Он использует *show_spot* (*p*) для отображения красной точки в точке *p* на экране через каждые полсекунды (значение, предопределенное для *Spot_time*). Для понимания деталей нам необходимы концепции, которые предстоит еще ввести по ходу обсуждения, так что пока просто посмотрите, как выглядит цикл.

```
from
    Line8.start
until
    Line8.is_after
loop
    show_spot (Line8.item.location)
    Line8.forth
end
```

Тело цикла

Цикл передвигает курсор (виртуальный маркер) к началу линии (*start*); затем, пока курсор не достиг последней позиции (*is_after*), он выполняет для каждой станции (*item*) следующие действия: отображает красное пятно в точке расположения станции – *location,* и передвигает курсор командой *forth* к следующей станции.

Каждое такое выполнение тела цикла называется *итерацией* цикла.

Рассмотрим ключевые составляющие цикла: инициализация (**from**), условие выхода (**until**) и повторно выполняемые действия (**loop**). Для получения полного понимания работы цикла следует предварительно проанализировать лежащие в его основе концепции.

Время программирования!

Анимация Линии 8

Поместите приведенный цикл в метод класса (пример класса для этой главы). Для этого примера и последующих вариаций обновите класс и запустите систему, чтобы увидеть результаты выполнения.

Цикл как аппроксимация

Как прием решения задач цикл является методом аппроксимации результата на последовательных, все расширяющихся подмножествах пространства задачи.

В примере с анимацией линии метро вся задача состоит в том, чтобы отобразить красную точку последовательно на каждой станции линии. Последовательными аппроксимациями являются: вначале точка не отображается ни на одной станции, затем отображается на первой станции, затем на второй, и так далее.

Вот еще один пример. Предположим, требуется узнать максимум множества из одного или нескольких значений *N*1*, N*2*, ..., N*n. Следующая стратегия неформально описывает эту работу.

I1 Положить *max* равным *N*1. После этого справедливо, что *max* является максимумом множества, состоящего ровно из одного значения, *N*1.

I2 Затем для каждого последовательного $i = 2, 3,..., n$ выполнять следующее: если *Ni* больше чем текущее значении *max*, переопределить *max*, сделав его равным *Ni*.

Из этого следует, что на *i-м* шаге (где первый шаг соответствует случаю I1, а последующие шаги – случаю I2 для $i = 2$, $i = 3$ и т.д.) действует следующее свойство, называемое **инвариантом цикла**.

Инвариант цикла стратегии «максимум» на шаге *i*

max является максимумом из $N_1, N_2, ..., N_i$

Этот инвариант для *n*-го шага, случай I2 для $i = n$, дает

```
"max является максимумом N1, N2, …, Nn",
```

что и является желаемым результатом.

Следующая картинка иллюстрирует стратегию цикла для этого случая.

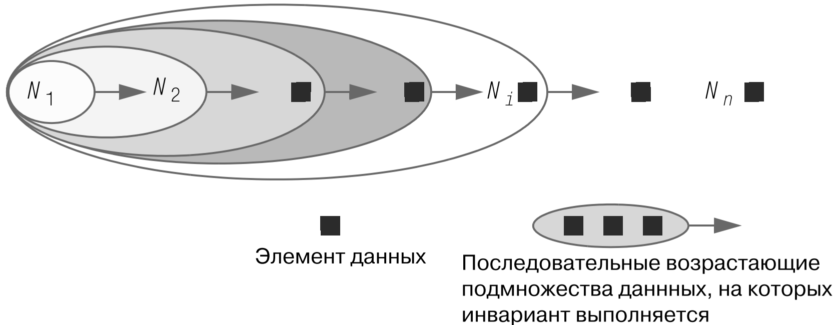


Рис. 7.5. Поиск максимума последовательными аппроксимациями

Цикл вначале устанавливает инвариант «*max* является максимумом первых *i* значений» для тривиального случая: $i = 1$. Затем повторно расширяется множество данных, на котором выполняется инвариант.

В примере с анимацией линии метро инвариантом является утверждение «красная точка отображалась на всех уже посещенных станциях».

Понятие инварианта не стало новым, так как мы уже знакомы с инвариантами класса. Две формы инварианта являются связанными, так как обе описывают свойство, которое некоторые операции должны сохранять. Но роли их различны: инвариант класса применим ко всему классу и должен сопровождать выполнение методов класса. Инвариант цикла применим к одному определенному циклу и должен сопровождать каждую итерацию цикла.

Стратегия цикла

Хотя большинство циклов более сложные, чем простой пример вычисления максимума, он иллюстрирует общую форму цикла как стратегию решения задач.

Эта стратегия полезна, когда задача сводится к следующей постановке. Мы начинаем с некоторого начального свойства *Pre*, и нам требуется достичь выполнения некоторой цели *Post*, характеризующей множество данных *DS*. Это множество конечно, хотя и может быть очень большим.

Чтобы использовать цикл, нужно найти слабую (более общую) форму цели *Post*: свойство *INV* (*s*) – инвариант цикла, определенный на подмножествах *s* из *DS* со следующими свойствами.

L1. Можно найти начальное подмножество *Init* из *DS*, такое, что начальное условие *Pre* влечет *INV* (*Init*); другими словами, инвариант выполняется на начальном подмножестве.

L2. *INV* (*DS*) влечет *Post* (из истинности инварианта на всем множестве следует истинность цели).

L3. Известен способ, позволяющий расширить подмножество *s*, для которого инвариант *INV* (*s*) выполняется, на большее подмножество (*s’*) из *DS*, сохраняя при этом истинность инварианта.

Пример с максимумом имеет все три составляющие: *DS* – это множество чисел$\{N_1, N_2, ..., N_n\}$; предусловие *Pre* говорит, что *DS* имеет, по меньшей мере, один элемент; цель *Post* утверждает, что найден максимум этих чисел, а инвариант *INV* (*s*), где *s* – подмножество $N_1, N_2, ..., N_i$, утверждает, что найден максимум на этом подмножестве. Тогда:

M1. Если *Pre* выполняется, (имеется хотя бы одно число), то нам известно начальное множество *Init*, такое, что *INV* (*Init*) выполняется. Для этого достаточно в качестве подмножества взять первый элемент N_1.

M2. *INV* (*DS*) – инвариант, применимый ко всему подмножеству $\{N_1, N_2, ..., N_n\}$ – влечет истинность цели, а фактически совпадает с целью *Post*.

M3. Когда *INV* (*s*) удовлетворяется на $s = \{N_1, N_2, ..., N_i\}$, которое пока не совпадает со всем *DS* – другими словами, $i < n$, – то мы можем установить *INV* (*s’*) для большего подмножества *s’* из *DS*: мы просто возьмем *s’* как $\{N_1, N_2, ..., N_i, N_{i+1}\}$, а в качестве нового максимума большее из двух чисел: старого максимума и N_{i+1}.

Заметьте: в общем случае нужно крайне тщательно проектировать инвариант, чтобы он позволял применить стратегию последовательной аппроксимации.

- *INV* должен быть достаточно **слабым**, чтобы его можно было применить к некоторому начальному подмножеству, обычно содержащему совсем немного элементов из всего множества.
- Он достаточно **силен**, чтобы из него следовала цель *Post*, когда он выполняется на всем множестве.
- Он достаточно **гибкий**, чтобы позволять расширять множество, сохраняя его истинность.

Начав с начального подмножества и повторяя расширения, мы придем к желаемому результату. Эта стратегия последовательной аппроксимации цели на прогрессивно растущих подмножествах может требовать большого числа итераций, но компьютеры быстры, и они не бастуют, требуя покончить с однообразной работой.

Эти наблюдения определяют, как работает цикл в качестве структуры управления. Его выполнение проходит следующие этапы.

X1. Устанавливается *INV* (*Init*), используя преимуществ L1. Это дает нам *Init* как первое подмножество *s*, на котором выполняется *INV*.

X2. До тех пор, пока *s* не совпадает со всем множеством *DS*, применяется прием L3 для установления *INV* на новом, расширенном *s*.

X3. Останавливаемся сразу же, когда *s* совпадает с *DS*. В этот момент *INV* выполняется на *DS*, что, благодаря L2, свидетельствует о достижении цели *Post*.

Этот процесс гарантирует завершаемость, поскольку мы предполагаем, что *DS* является конечным множеством, *s* всегда остается подмножеством *DS* и на каждом шаге увеличивается по крайней мере на один элемент, так что гарантируется, что после конечного числа шагов *s* совпадет с *DS*. В некоторых случаях, однако, установление завершаемости не столь просто.

Оператор цикла: базисный синтаксис

Для выражения стратегии цикла в тексте программы будем использовать для нашего примера «максимума» следующую общую структуру:

```
from
    –"Определить max равным N_1"
    –"Определить i равным 1"
until
    i = n
loop
    –"Определить max как максимум из {max, N_i+1}"
    –"Увеличить i на 1"
end
```

На данный момент все составляющие операторы являются псевдокодом. Пример демонстрирует три требуемые части конструкции цикла (дополненные позже двумя *возможными* частями).

- Предложение **from** вводит операторы инициализации (X1).
- Предложение **loop** вводит операторы, выполняемые на каждой последовательной итерации (X2).
- Предложение **until** вводит условие выхода – условие, при котором итерации завершаются (X3).

Эффект выполнения этой конструкции, задаваемой ключевыми словами(**from**, **until**, **loop**), определяется в соответствии с предыдущим обсуждением.

- Первым делом выполняются операторы предложения **from** (*инициализация*).
- Пока не выполняется *условие выхода* в предложении **until**, выполняются операторы предложения **loop** (*тело цикла*).

Более точно это означает, что после инициализации тело цикла может быть выполнено:

- ни разу, если условие выхода из цикла выполняется сразу же после инициализации;

- один раз, если выполнение тела цикла приводит к выполнению условия выхода;
- общая ситуация: i раз для некоторого i, если условие выхода будет ложным для всех j выполнений тела цикла $1 \leqslant j < i$ и становится истинным после i-го выполнения.

Синтаксически предложения **from** и **loop** каждое содержат составной оператор. Как следствие, здесь можно использовать произвольное число операторов, в том числе и ноль. Это происходит, например, если цикл не нуждается в явной инициализации (в тех случаях, когда контекст, предшествующий циклу, обеспечивает истинность инварианта цикла); тогда предложение **from** будет пусто.

```
From

            <-------------- Здесь ничего

until
       –"Условие выхода"
loop
       – "Тело цикла"
end
```

В теле цикла должно быть некое продвижение в процессе аппроксимации (добавить хотя бы один новый элемент к подмножеству предыдущего обсуждения); в противном случае процесс никогда не завершится. Поэтому в реальных программах никогда не встречается цикл с пустым предложением **loop**.

Включение инварианта

Базисная форма цикла, как мы видели, не показывает инвариант цикла. А жаль, так как инвариант является основой для понимания того, что цикл делает. Поэтому рекомендуемая и возможная форма записи цикла включает предложение **invariant**. С данным предложением цикл выглядит так:

```
from
    –"Определить max равным N1"
    –"Определить i равным 1"
invariant
    –"max является максимумом подмножества {N1, N2, … Ni}"
until
    i = n
loop
    –"Определить max как максимум из {max, Ni+1}"
    –"Увеличить i на 1"
end
```

Инвариант в этом примере – все еще псевдокод, но тем не менее, он полезен, так как поставляет важную информацию о цикле.

Оператор цикла: корректность

Инвариант цикла имеет два важных характеристических свойства.

Корректность

> **Принцип инварианта цикла**
> Инвариант цикла должен:
> I1 стать истинным после инициализации (предложения **from**);
> I2 оставаться истинным всякий раз после выполнения тела цикла (предложения **loop**) в предположении, что значение условия выхода было ложным.

Оператор цикла в результате своего выполнения сохраняет свойство, заданное инвариантом, начиная с удовлетворения свойства, сохраняя свойство после каждой итерации, завершая работу с выполняемым свойством. Это сохранение свойства объясняет полученное им имя «инвариант», применимое здесь к инвариантам цикла (как к ранее инвариантам класса, которые подобным образом должны стать истинными в результате выполнения процедуры создания и оставаться истинными после каждого выполнения метода класса).

Как мы уже видели, цель цикла – в достижении определенного результата путем последовательных аппроксимаций. Шагами, направленными на достижение цели, являются инициализация и последовательное выполнение тела цикла. После каждого из этих шагов свойство устанавливает, что аппроксимация приближает финальный желаемый результат. Инвариант определяет аппроксимацию. В наших двух примерах инвариантами являются:

- «*max* является максимумом $\{N_1, N_2, ..., N_i\}$», как i-я аппроксимация, для $1 \leqslant i \leqslant n$, финального свойства «*max* является максимумом $\{N_1, N_2, ..., N_n\}$»;
- «красная точка отображается в порядке следования станций, посещенных на линии до текущего момента», как аппроксимация заключительного свойства, что точка отображается на всех станциях линии.

Пусть *Loop_invariant* будет инвариантом. Когда выполнение цикла завершится, инвариант все еще будет выполняться по свойствам I1 и I2 принципа инварианта цикла. Кроме того, условие выхода из цикла *Loop_exit* также существует, так как в противном случае цикл никогда бы не завершился. Поэтому заключительным условием, истинным в момент завершения, является конъюнкция:

```
Loop_invariant and Loop_exit
```

Это условие и определяет результат выполнения цикла.

Корректность

> **Принцип постусловия цикла**
> Условие, достижимое в результате выполнения цикла, представляет конъюнкцию инварианта и условия выхода.

Синтаксис подчеркивает эту связь – принцип постусловия цикла, – размещая предложения **invariant** и **until** друг за другом. Так что, если вы видите инвариант цикла и хотите знать, каков достигнут результат, просто посмотрите на два рядом идущих условия:

```
from
    -- "Определить max равным N1"
    -- "Определить i равным 1"
invariant
    "max is the maximum of {N1, N2, ..., Ni}"
until
    i = n
loop
    -- "Определить max как максимум из {max, Ni+1}"
    -- "Увеличить i на 1"
end
```

and — Заключительное условие

В примере с линией метро неформально условие выхода говорит: «все станции посещены». Конъюнкция этого условия с инвариантом говорит нам, что красная точка отображалась на всех станциях.

Принцип постусловия цикла является прямым следствием определения цикла как механизма аппроксимации. Цитируя предыдущее обсуждение, напомним, что идея состояла в выборе инварианта как некоторого обобщения цели:

- «достаточно **слабого**, чтобы его можно было применить к некоторому начальному подмножеству всего множества»: в этом роль инициализации;
- «достаточно **гибкого**, чтобы позволять расширять множество, сохраняя его истинность»: в этом роль тела цикла, выполняемого при истинности инварианта и ложности условия выхода из цикла. Тело цикла должно обеспечить расширение множества и сохранение инварианта;
- «достаточно **сильного**, чтобы из него следовала цель *Post*, когда он выполняется на всем множестве»: это достигается на выходе, когда в соответствии с принципом постусловия цикла становится истинной конъюнкция условия выхода и инварианта.

Завершение цикла и проблема остановки

Согласно описанной схеме выполнения цикла тело цикла выполняется до тех пор, пока не будет выполнено условие выхода. Если цикл строится в соответствии с рассмотренной стратегией аппроксимации, то его выполнение завершится, так как рассматривается конечное множество данных, на каждом шаге происходит расширение подмножества хотя бы на один элемент, потому за конечное число шагов процесс обязательно завершится. Но синтаксис цикла допускает произвольную инициализацию, произвольное тело цикла и условие выхода, так что цикл может никогда не завершиться, как в этом примере:

```
from
    –"Любой оператор (возможно, пусто)"
until
    0 /= 0
loop
    –"Любой оператор (возможно, пусто)"
end
```

В этом искусственном, но допустимом примере, условие цикла – которое, конечно, можно просто записать как **False**, – никогда не выполнится, так что цикл не может завершиться. Если запустить эту программу на компьютере, можно долго сидеть у экрана, где

ничего не происходит, хотя компьютер работает. Осознав, что происходит что-то неладное, вы придете к решению, что нужно прервать выполнение (в EiffelStudio для этого есть специальная кнопка). Но у вас нет способа узнать – если вы пользователь программы и не имеете доступа к ее тексту, – «зациклилась» ли программа или просто идут долгие вычисления.

Если программа не завершается, то, хочу напомнить, это еще не является свидетельством ошибочного поведения. Некоторые программы проектируются специально так, чтобы они всегда работали, пока не будут явно остановлены. Прекрасным примером является операционная система (ОС), установленная на компьютере. Я был бы весьма разочарован, если бы при наборе этого текста произошло завершение работы ОС, которое бы означало, что система «рухнула» или я случайно нажал ногой кнопку выключения компьютера. Такая же ситуация существует с большинством «встроенных систем» (программ, выполняемых на различных устройствах): вы не захотите завершения программы мобильного телефона во время вашего разговора.

Но с другой стороны, обычные программы – в частности, большинство программ, обсуждаемых в этой книге, – устроены так, что, получив некоторый ввод, они вырабатывают результат за конечное число шагов.

Когда пишутся такие программы, можно непреднамеренно построить цикл, условие выхода из которого не будет выполняться, что приведет к тому, что вся программа не завершится. Чтобы избежать этого неприятного результата, лучший способ, позволяющий убедиться, что цикл завершится, состоит во введении для каждого цикла элемента, называемого **вариантом цикла**.

Определение: Вариант цикла

Вариантом цикла является целочисленное выражение, обладающее тремя свойствами.

V1 После выполнения инициализации (предложения **from**) вариант получает неотрицательное значение.

V2 Каждое выполнение тела цикла (предложения **loop**), когда условие выхода не выполняется, а инвариант выполняется, уменьшает значение варианта.

V3 Каждое такое выполнение оставляет вариант неотрицательным.

Если удается построить вариант, то тем самым доказывается завершаемость цикла за конечное число итераций, поскольку невозможно для неотрицательного целого значения уменьшаться бесконечно, оставаясь неотрицательным. Если мы знаем, что после инициализации вариант получил значение V, то после максимум V итераций цикл завершится, так как каждая итерация уменьшает вариант минимум на 1.

В этом доказательстве существенно, что вариант имеет целочисленный тип. Вещественный тип не годится, поскольку можно построить бесконечную уменьшающуюся последовательность, такую как 1, 1/2, 1/3, …, $1/n$, … (это строго верно в математике; на компьютере, где бесконечности нет, множество закончилось бы, достигнув «машинного» нуля).

Если вы знаете вариант цикла, синтаксис позволяет вам указать его в предложении **variant**, следующем после тела **цикла**. Например, мы можем добавить спецификацию варианта цикла в наше вычисление максимума:

```
from
    –"Определить max равным N1"
    –"Определить i равным 1"
invariant
    1 <= i
    i <= n
    – "max является максимумом подмножества {N1, N2, …, Ni }"
until
    i = n
loop
    –"Определить max как максимум из {max, Ni+1}"
    –"Увеличить i на 1"
variant
    n - i
end
```

Вариантом является выражение $n - i$. Он удовлетворяет требуемым условиям.

V1 Инициализация устанавливает i в 1. Предусловие цикла предполагает $n \geqslant 1$. Так что вариант после инициализации неотрицателен.

V2 Тело цикла увеличивает i на единицу, соответственно уменьшая вариант.

V3 Если условие выхода не выполняется, то i будет меньше n ($i < n$, а не $i <= n$) и, следовательно, $n - i$, будучи уменьшенным на единицу, останется неотрицательным.

Из условия V3 следует, что недостаточно рассматривать отрицание условия выхода, которое говорит нам только, что $i /= n$: нам нужно быть уверенным, что $i < n$. Заметьте, для этого добавлены новые свойства в инвариант цикла: $1 <= i$ и $i <= n$. Они выполняются после инициализации цикла и сохраняются при выполнении очередной итерации. Поэтому, когда условие выхода не выполняется, $i /= n$, то с учетом инварианта $i <= n$ фактически $i < n$.

Возможно, вы полагаете, глядя на этот пример, что много шума из ничего: и так понятно, что программа корректна и всегда завершится. Но на практике довольно часто встречаются программы, которые «зависают» по непонятным для пользователя причинам, и это означает, что в программе есть циклы, которые не завершаются. Возможно, что такая ошибка не обнаруживается на тестах, созданных разработчиком программы, поскольку тесты охватывают лишь малую часть возможных случаев и не всегда учитывают фантазию пользователя. Только благодаря выводу, проиллюстрированному выше, можно гарантировать – в вашей собственной программе – что цикл *всегда* завершится, вне зависимости от ввода.

Рассмотрение возможности незавершаемости программ приводит к введению важных понятий, изучаемых в отдельном курсе по теории вычислений.

Почувствуй теорию

Проблема остановки и неразрешимость

Возможность существования в программе бесконечно работающих циклов вызывает тревогу. Нет ли автоматического способа проверки, который мог бы для каждой данной программы сказать, завершаются ли в ней все циклы или нет? Мы знаем, что компиляторы делают для нас многие проверки, в частности, проверку типов (если *x* типа *STATION* и встречается вызов *x.f*, то компилятор откажется от компиляции программы и выдаст сообщение об

ошибке, если *f* не является методом класса *STATION*). Может быть, компилятор способен выполнять проверку завершаемости цикла?

Ответ на этот вопрос отрицателен. Известная теорема говорит, что если язык программирования достаточно мощный для практических потребностей, то *невозможно* написать программу, такую как компилятор, которая могла бы корректно отвечать на вопрос, будет ли предъявленная программа всегда завершаться, анализируя поданный ей на вход программный текст. Этот результат известен как **неразрешимость Проблемы Остановки.**

- *Проблема Остановки* – будет ли программа завершаться (останавливаться).
- Проблема *неразрешима,* если нет эффективного алгоритма, вырабатывающего корректное решение в каждом случае.

Проблема Остановки является одним из наиболее известных результатов о неразрешимости в теории вычислений, хотя далеко не единственным.

Хотя результат о неразрешимости может настраивать на пессимистический лад, он вовсе не означает, что когда вы пишете программу, не требуется заботиться о гарантиях ее завершаемости. Теорема о неразрешимости устанавливает отсутствие *общего* автоматического механизма, который бы устанавливал завершаемость *любой* программы, но это не означает, что нет способов установления завершаемости для *некоторых* программ. Вариант цикла – это пример такого весьма эффективного приема. Если вы для вашего цикла сможете доказать существование целочисленного выражения со свойствами V1-V3, то можете гарантировать, что цикл завершится.

Существующие коммерческие компиляторы все же не способны выстраивать такие доказательства, так что их нужно проводить самостоятельно, анализируя текст программы, и если есть какие-либо сомнения, то позвольте EiffelStudio проверять в момент выполнения, что вариант уменьшается на каждой итерации. В отличие от общей Проблемы Остановки отсутствие автоматического доказательства свойств варианта не представляет фундаментальную невозможность, а является ограничением текущей технологии.

Мы будем иметь возможность доказать утверждение о неразрешимости Проблемы Остановки сразу же после изучения подпрограмм в следующей главе.

Почувствуй историю:

Поиски решения Проблемы Остановки

Проблема Остановки была описана как специальный случай «проблемы разрешимости» или *Entscheidungsproblem,* вопрос, восходящий к Лейбницу в 17-18 веках и к Гильберту в начале 20-го века. Его неразрешимость была доказана за десятилетие до появления настоящих компьютеров с хранимой памятью в знаменитой математической работе в 1936 году «*On Computable Numbers*, *with an Application to the Entscheidungsproblem*» («О вычислимых числах с приложением к проблеме разрешимости»).

Автор, британский математик Алан Тьюринг, для доказательства ввел абстрактную модель вычисления, известную сегодня как машина Тьюринга. Машина Тьюринга – математическая концепция, а не физическое устройство – активно используется при обсуждении общих свойств вычислений, независимо от архитектуры компьютера или языка программирования.

Тьюринг не остановился на работе с чисто математическими машинами. Во время Второй мировой войны он принимал активное участие в дешифровании немецких сообщений, зашифрованных с использованием криптографической машины Enigma. После войны участвовал в строительстве первых настоящих компьютеров (конец его жизни был омрачен – позвольте оставаться вежливым – недостаточным признанием его заслуг официальными лицами его страны).
Алан Тьюринг ввел много плодотворных идей в компьютерные науки. Высочайшим признанием достижений в этой области является премия Тьюринга, установленная в его честь.

Неразрешимость Проблемы Остановки относится к серии *отрицательных* результатов, потрясавших одну область науки за другой, разрушая великие научные торжества, которые новое столетие поторопилось провозгласить.

- Математики, видя правильность теории множеств и основываясь на базисных методах логического вывода, смогли справиться с появлением видимых *парадоксов*, а затем, благодаря 30-летним усилиям Бертрана Рассела и Давида Гильберта по восстановлению основ математики, казалось, могли рассчитывать на успех. Но Курт Гёдель доказал, что в любой аксиоматической системе, достаточно мощной, чтобы описать обычную математику, всегда найдутся утверждения, которые нельзя ни доказать, ни опровергнуть. **Теорема о неполноте** является одним из наиболее поразительных примеров ограничений наших способностей к *выводу*.
- Примерно в то же время физикам пришлось принять принцип неопределенности Гейзенберга и другие результаты квантовой теории, ограничивающие наши способности наблюдения.

Результаты о неразрешимости, в частности, проблема остановки, являются компьютерными версиями таких, по-видимому, абсолютных ограничений.

Теоретическая неразрешимость Проблемы Остановки не должна оказать прямого воздействия на вашу практику программирования – за исключением эмоциональной травмы от осознания наших интеллектуальных ограничений, но я верю, что вы сумеете справиться с этим. Однако же программы, которые не завершаются, не просто теоретическая возможность – это реальная опасность. Чтобы избежать их неприятного появления, помните:

Почувствуй методологию

Завершение цикла

Всякий раз, когда вы пишете цикл, исследуйте вопрос его завершения. Убедите себя, предложив подходящий вариант, что цикл всегда будет выполнять конечное число итераций. Если не удается, попробуйте переделать цикл так, чтобы вы могли поставлять его с вариантом.

Анимация линии метро

В качестве простого примера цикла вернемся к задаче, в общих чертах рассмотренной в начале этой главы, — анимации Линии 8 метро, устанавливая красную точку при посещении очередной станции. Мы можем использовать:

- из класса *STATION* запрос *location*, определяющий положение станции на карте. Результат запроса принадлежит типу *POINT*, представляющего точку в двумерном пространстве;
- команду *show_spot* из класса *TOURISM*, имеющую аргумент *show_spot* (*p*), где *p* типа *POINT*. Команда отображает красную точку в положении *p*;
- команду *spot_time* также из класса *TOURISM*, с предопределенным значением времени, в течение которого красная точка остается на каждой станции; сейчас оно имеет значение 0,5 секунды.

Задача цикла состоит в том, чтобы последовательно вызывать метод *show_spot*, передавая методу расположение каждой станции на линии.

Для последовательного получения станций мы можем (с помощью операций над переменными, изучаемыми в 9-й главе) использовать запрос *i_th*, который дает нам *i*-й элемент линии при вызове *some_line.i_th* (*i*) для любого применимого *i*. Цикл должен выполнять

```
show_spot (Line8.i_th (i ).location)
```

для последовательных значений *i*, в пределах от 1 до *Line8.count*. Позвольте вместо использования этой возможности познакомить вас с типичной формой цикла, которая выполняет итерации над специальной структурой объектов, называемой списком. «*Итерирование*» структуры означает выполнение операции над каждым ее элементом или на некотором подмножестве элементов, заданном явным критерием. В данном примере операция состоит в вызове метода *show_spot* в точке расположения каждой выбранной станции.

Классы, такие как *LINE*, вообще классы, описывающие упорядоченные списки объектов, поддерживают итерацию, позволяя передвигать курсор (специальную метку) к следующим элементам списка. Курсор не должен быть фактическим объектом, это абстрактное понятие, обозначающее в каждый конкретный момент позицию в списке.

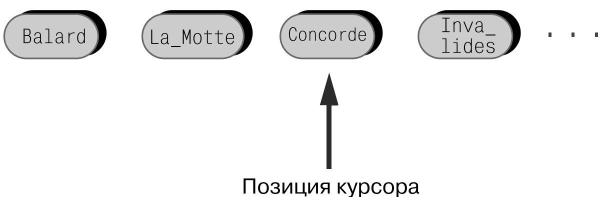


Рис. 7.6. Список и его курсор

В показанном на рисунке состоянии курсор на третьей станции. Класс и другие классы, задающие списки, включают четыре ключевых метода – две команды и два запроса – для итерирования соответствующей структуры объектов:

- команду *start*, которая устанавливает курсор на первом элементе списка (в нашем примере элементом является станция метро);
- команду *forth*, которая передвигает курсор к следующему элементу;
- запрос *item*, возвращающий элемент списка, если он существует в позиции курсора;
- булевский запрос *is_after*, возвращающий **True**, если и только если курсор находится в позиции справа от последнего элемента (после списка). Для симметрии есть запрос *is_before*, хотя он нам пока не понадобится.

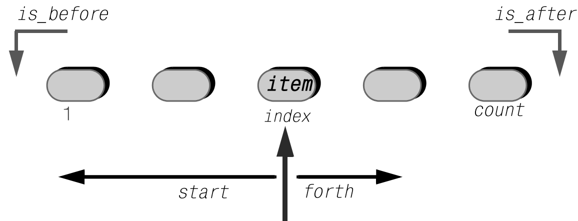


Рис. 7.7. Список основных методов

Полезен также запрос *index*, который, как показано, дает индекс текущей позиции курсора. Предполагается, что индекс позиции первого элемента равен 1 и равен *count* для последнего элемента.

Этого достаточно для построения общей схемы итерирования списка и ее применения в нашем случае:

```
from
    Line8 .start
invariant
– "Точка отображена для всех станций перед позицией курсора"
– "Другие утверждения, задающие инвариант (смотри ниже)"
until
    Line8 .is_after
loop
    show_spot (Line8 .item .location)
    Line8 .forth
variant
    Line8 .count - Line8 .index + 1
end
```

Эта схема, использующая *start*, *forth*, *item* и *is_after* для итерирования списка, столь часто встречается, что следует понимать все ее детали и быть в полной уверенности ее корректности. Неформально ее эффект ясен.

- Установить курсор на первый элемент списка, если он есть, вызвав *start* в разделе инициализации.
- На каждом шаге цикла в течение *Spot_time* секунд отображать точку на станции *Line8.item* в позиции курсора.
- После отображения точки на каждом шаге цикла передвинуть курсор на одну позицию, используя *forth*.
- Закончить цикл, когда курсор в позиции *is_after*, после последнего элемента.

Во избежание любых недоразумений (а я надеюсь, что предыдущее обсуждение не оставило им места) напомним – нет никакой связи между позицией станции на карте, ее **местоположением** и понятием **позиции курсора:**

- станция имеет географическое положение в городе, определяемое ее координатами;
- курсор существует только в нашем воображении и в нашей программе, но не во внешнем мире. Это внутренняя метка, позволяющая программе выполнять итерирование списка, запоминая от одного итерационного шага до другого, какой элемент был посещен последним.

Время программирования!

Завершающиеся и незавершающиеся циклы

Обновите цикл в методе *traverse* класса *ROUTES* в соответствии с последней версией с вариантом и (неформальным) инвариантом. Выполните систему.

Теперь удалите оператор *Line8.forth*, введя ошибку. Снова запустите систему и посмотрите, что получится (затем восстановите удаленную строку для дальнейших упражнений).

Давайте рассмотрим составляющие цикла детальнее: инициализация использует *start*, чтобы установит курсор в первую позицию. В облике контракта класса *LINE* (и в любом другом классе, построенном на понятии списка) можно видеть, что спецификация *start* говорит:

```
start
    – Установить курсор на первом элементе
    – (Не имеет эффекта, если список пуст)
    ensure
    at_first: (not is_empty) implies (index = 1)
    empty_convention: is_empty implies is_after
```

Булевский запрос *is_empty* определяет, пуст ли список. В данный момент рассмотрим только случай не пустого списка (подобного *Line8*). Первое предложение at_first постусловия для *start* устанавливает, что после инициализации курсор указывает на первый элемент, (*index* = 1), как нам и требуется.

Условие выхода из цикла – *Line8.is_after*. Для непустого списка оно не будет выполняться после инициализации: это нетрудно установить, анализируя инвариант класса, который говорит:

```
is_after = (index = count + 1)
```

Эквивалентность двух булевских значений означает, что *is_after* истинно, если и только если *index* = *count* + 1; для не пустого списка *count* будет, по крайней мере, равно 1, так что после инициализации, когда *index* = 1, невозможно выполнение *is_after*. В этом случае тело цикла будет выполнено, по крайней мере, один раз.

На каждом шаге тела цикла выполняется

```
show_spot (Line8.item.location)
```

В результате отображается красная точка в географическом местоположении элемента, на который указывает курсор.

Понимание и верификация цикла

Давайте добьемся более глубокого понимания цикла в нашем примере путем верификации его корректности. Проверим, что он выполняется во всех случаях в соответствии с ожиданиями. Хорошая идея – использовать отладчик, анализируя различные состояния объектов во время выполнения программы.

Время программирования!

Используйте отладчик

Теперь, когда вы ознакомились с пояснениями примера, в частности, с аргументами корректности, полезно взглянуть на конкретную картину того, что происходит во время выполнения. Эту возможность обеспечивает отладчик EiffelStudio. Используйте его как для выполнения программы в целом, так и пошагово, оператор за оператором, при выполнении метода *traverse* класса *ROUTES*. В этом случае можно остановиться в любой момент и проанализировать содержимое все интересующих вас объектов.

Например, можно видеть экземпляр *LINE* и проверить результаты запросов, таких как *is_before* и *is_after*, согласуются ли они с ожидаемыми значениями, выведенными из анализа программы.

Такой инструментарий времени выполнения не является заменой обоснования свойств программы. Выводы дают вам свойства, верные при всех выполнениях программы, в то время как проверка периода выполнения говорит нам о том, что свойство выполняется в данном конкретном запуске программы. Но все же отладка крайне полезна как способ получения преимуществ от практического понимания того, что делается: она позволяет вам буквально *видеть*, как программа работает.

В полном соответствии со своим именем отладчик помогает, когда программа выполняется не так, как ожидалось: найти ошибку – выловить «жучка». Но область его применения шире – жучок или не жучок, но это дает вам окно наблюдения за процессом выполнения программы. Не ждите, пока что-то случится, пользуйтесь преимуществами, предоставляемыми отладчиком.

Раздел приложения EiffelStudio подскажет вам, как запустить отладчик.

Вернемся к нашему циклу:

```
from                                                              [5]
    Line8 .start
invariant
    – "Точка отображена для всех станций перед позицией курсора"
    – "Другие утверждения, задающие инвариант (смотри ниже)"
until
    Line8 .is_after
loop
    show_spot (Line8 .item .location)
    Line8 .forth
variant
    Line8 .count - Line8 .index + 1
end
```

Начнем анализ со случая пустого списка. Как отмечалось, постусловие команды *start* говорит:

```
ensure
    at_first: (not is_empty) implies (index = 1)
    empty_convention: is_empty implies is_after
```

В соответствии с соглашением на пустом списке после вызова *start* выполняется запрос *is_after*. Конечно, это «соглашение» не случайно – оно в точности соответствует нашим намерениям при создании типовой схемы итерации списка: начинать со *start*, выходить по *is_after* и каждый раз, выполнив операцию над элементом *item*, перейти к следующему элементу *forth*. Когда эта схема применяется к пустому списку, она не должна производить никакого видимого эффекта и должна останавливаться незамедлительно.

Почувствуй методологию:

Заботьтесь о граничных случаях!
Экстремальные случаи, такие как пустой список, являются частой причиной ошибок. Достаточно просто при проектировании программы думать о настоящих полных списках (и при отладке рассматривать только эти случаи). Когда же программа при одном из выполнений встречается с пустой структурой – она ломается. Это случается не только с пустыми структурами, но, в целом, с другими экстремальными случаями. Например, когда мы имеем дело со структурами ограниченного размера, экстремальным является случай, когда достигается предельный размер.
При проектировании программы и обосновании ее корректности убедитесь, что вы рассматриваете и граничные случаи. Это же верно и при тестировании – всегда включайте граничные случаи в систему тестов.

Вы можете запустить пример на пустой линии (класс *TOURISM* определяет метод *Empty_line* для этих целей) и использовать отладчик, чтобы посмотреть на результат.

Итак, цикл правильно себя ведет на пустом списке. В оставшейся части обсуждения корректности будем полагать, что список не пуст.

Рассмотрим спецификацию для *item*. Из предусловия ясно, что запрос применим, если курсор указывает на элемент списка:

```
item: STATION
        -- Текущий элемент
    require
        not_off: not (is_after or is_before)
```

Этот факт отражает рисунок:

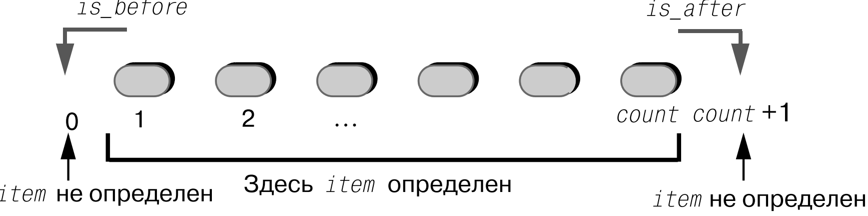


Рис. 7.8. Где определены элементы списка

Так как в теле цикла вызывается *item* – в вызове *show_spot* – следует убедиться, что перед выполнением вызова предусловие всегда будет выполняться.

Заметьте, условием выхода является **not** *is_after*, так что *is_after* не выполняется, когда вызывается *show_spot* (если бы оно выполнялось, то тело цикла не выполнялось бы). Кроме того, *is_before* также не будет выполняться. Мы можем добавить следующее свойство в инвариант цикла:

```
not_before_unless_empty: (not is_empty) implies (not is_before)          [6]
```

Давайте проверим, что это на самом деле инвариантное свойство. Как отмечалось, мы рассматриваем только непустые списки (если список пуст, то **[6]** выполняется тривиально), так что нам нужно проверить, что **not** *is_before* удовлетворяет свойству инвариантности цикла. В ***инварианте класса*** мы установили, что:

```
is_before = (index = 0)
index >= 0
index <= count + 1
```

Другими словами, *is_before* истинно, если и только если индекс позиции курсора равен нулю. После инициализации постусловие – предложение *at_first*, приведенное выше, – говорит, что индекс равен 1, так что *is_before* равно **False** и выполняется **not** *is_before*. Спецификация *forth* устанавливает:

```
forth
        – Передвинуть курсор к следующей позиции
        require
            not_after: not is_after
        ensure
            moved_forth: index = old index + 1
```

Так как *index* никогда не бывает отрицательным и увеличивается на каждом шаге на 1, то он не может быть нулем, следовательно, *is_before* не может иметь места. Так что [6] действительно инвариантное свойство.

Полезно выполнить трассировку выполнения цикла; используйте отладчик для выполнения цикла итерацией за итерацией, анализируя структуру объектов на каждом шаге.

Курсор – где он может находиться

Чтобы дополнить наше понимание цикла и примера, полезно вновь обратиться к инварианту класса *LINE*. Вы увидите два предложения, которые имеются у всех библиотечных классов, связанных со списковыми структурами:

```
non_negative_index: index >= 0
index_small_enough: index <= count + 1
```

Это означает, что мы разрешаем курсору быть установленным:

- на элементе списка, если таковые имеются;
- непосредственно слева от первого элемента, но не более чем на одну позицию;
- непосредственно справа от последнего элемента, но не более чем на одну позицию.

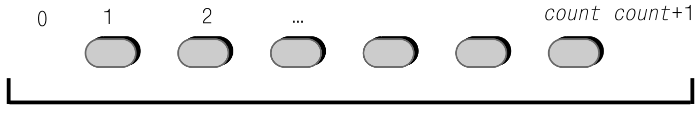


Рис. 7.9. Допустимые позиции курсора

Для общей схемы итерирования списка полезно то, что курсор может находиться на одной позиции слева или справа от элементов списка. Это иллюстрирует наш пример:

```
from
    some_list .start
invariant
    – "Все элементы левее курсора обработаны"
until
    some_list .is_after
loop
    – "Обработка item в позиции курсора"
    some_list .forth
variant
    some_list .count - some_list .index + 1
end
```

После обработки последнего элемента вызов *forth* передвинет курсор и *is_after* станет истинным, что гарантирует прекращение итераций. Что произойдет, когда в этой позиции (*count* + 1) вызвать *forth*, хотя здесь нет элементов списка? Предусловие метода *forth* запрещает такой вызов:

```
require
    not_after: not is_after
```

Этим наблюдением мы завершаем обзор основных свойств циклов и способов, позволяющих убедиться в их корректности.

7.6. Условные операторы

Следующая структура управления – условный оператор – не вызывает столько вопросов как цикл, но также относится к фундаментальным строительным блокам программы.

Условный оператор включает условие (в базисной форме) и два оператора. Он будет выполнять один из операторов в зависимости от выполнения условия.

Как способ решения задач, условный оператор соответствует *разделению вариантов*: разделить пространство задачи на две (или более) части, такие что проще решить задачу независимо для каждой части. Например, пусть мы пытаемся добраться от Эйфелевой башни до Лувра.

- *Если* погода хороша и мы не слишком устали, *то* дойти пешком до ближайшей станции метро, а далее добираться на метро.
- *Иначе* попытаться поймать такси.

Или классический пример из школьной математики: пусть нам нужно найти вещественные корни квадратного уравнения $ax^2 + bx + c = 0$.

- Если дискриминант d, определенный как $b^2 - 4ac$, положителен, то можно получить два решения $(-b \pm \sqrt{d}) / 2a$.
- Иначе если d равно нулю, то решение одно — $-b / 2a$.
- Иначе нет вещественных корней (только комплексные).

Рисунок представляет стратегию решения задачи путем разделения ее на области, определяемые условием:

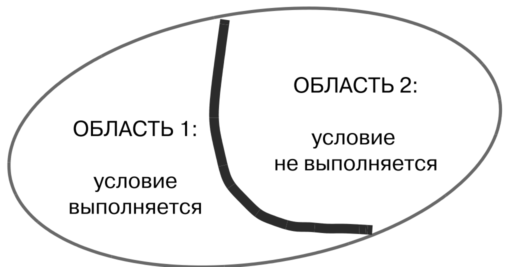


Рис. 7.10. Условие как способ разделения пространства задачи

Базисной формой конструкции, реализующей эту стратегию решения задач, является:

```
if условие then
    "Получить решение в Области 1"
else
    "Получить решение в Области 2"
end
```

Вскоре мы обобщим конструкцию на случай разделения на большее число вариантов.

Условный оператор: пример

Усовершенствуем наш последний пример с циклом. Представьте, что мы хотим, чтобы на пересадочных станциях вместо красной точки отображалась бы желтая мигающая точка в течение более длительного времени. Класс *TOURISM* предусмотрительно заготовил для этих целей метод *show_blinking_spot*, который дополняет метод *show_stop*, используемый ранее.

Для достижения требуемого результата в нашей программе придется сделать следующее изменение:

```
from                                                                   [7]
    Line8 .start
invariant
    not_before_unless_empty: (not is_empty) implies (not is_before)
    – "Точка отображена для всех станций перед позицией курсора"
until
    Line8 .is_after
loop
    if Line8 .item .is_exchange then
```

```
            show_blinking_spot (Line8 .item .location)
        else
            show_spot (Line8 .item .location)
        end
        Line8 .forth
    variant
        Line8 .count - Line8 .index + 1
    end
```

В условном операторе трижды используется выражение *Line8.item* – вызов запроса. Более элегантно вычислять такой результат один раз, дав ему имя, а затем использовать это имя. Вскоре мы покажем, как это делать.

Время программирования!

Используйте условный оператор
Измените предыдущий пример – метод *traverse* в классе *ROUTES* – с учетом приведенного выше условного оператора. Запустите систему и получите результат.

Для записи условного оператора нам понадобятся четыре новых ключевых слова: **if**, **then** и **else**, а также **elseif**, которое появится чуть позже. Базисная структура оператора перед вами:

```
if condition then
    Compound_1
else
    Compound_2
end
```

Здесь *condition* (условие) – это булевское выражение, а *Compound_1* и *Compound_2* – составные операторы, последовательности из нуля или более операторов.

Структура условного оператора и ее вариации

Будучи последовательностями из нуля и более операторов, как *Compound_1*, так и *Compound_2* могут быть пустыми операторами, поэтому можно писать (хотя такой стиль не рекомендуется):

```
if condition then
    Compound_1
else
                        <—— Здесь ничего
end
```

В этом варианте **else**-часть, хотя и присутствует, но в ней ничего нет. Это соответствует частому случаю, когда нужно выполнить некоторые действия при выполнении определенного условия; в противном случае делать ничего не нужно. Вместо того чтобы писать **else**-часть без операторов, разрешается просто ее опускать. Можно писать (рекомендуемый стиль):

```
if condition then
    Compound_1
                    <——— Без предложения else
end
```

В любой форме – с присутствием **else** или без него – любые операторы, входящие в составной оператор, могут быть сами структурами управления, циклами или составными операторами.

Предположим, что некоторые станции метро могут быть связаны с железной дорогой, и для таких станций нужно выполнять особые действия. Предлагаемую схему можно использовать вместо предыдущего цикла.

Стиль этого примера не рекомендуется. Рекомендуемый стиль записи появится чуть позже.

```
from … invariant … until … loop                                          [8]
                    – Пропущенные предложения цикла смотри в программе [7]
    if Line8 .item .is_exchange then
        show_blinking_spot (Line8 .item .location)
    else
        if Line8 .item .is_railway_connection then
            show_big_blue_spot (Line8 .item .location)
        else
            show_spot (Line8 .item .location)
        end
    end
    Line8 .forth
variant … end
```

Включение структур управления внутрь других структур называется гнездованием или вложенностью. Здесь условный оператор вложен в другой условный оператор, в свою очередь, вложенный в цикл.

Почувствуйте стиль

Какой может быть глубина вложенности?

Теоретически ограничений на глубину вложенности нет. Ограничения диктуются практикой: хорошим вкусом и желанием сохранить читабельность программы, простоту ее понимания.

В последнем примере **[8]** глубина вложенности равна четырем, и это максимум того, что следует допускать в повседневном программировании. Это не абсолютное правило: некоторые алгоритмы по-настоящему требуют более высокой вложенности. Но, всякий раз, когда вы достигаете такого уровня вложенности, следует остановиться и проанализировать, нельзя ли этого избежать.

Альтернативой обычно является выделение части структуры как независимой *подпрограммы*, с заменой непосредственного вхождения ее *вызовом*. В следующей главе о подпрограммах пойдет подробный разговор

В примерах, таких как **[8]**, глубина вложенности делает структуру более сложной, чем это фактически необходимо. Ее можно упростить, не обращаясь к подпрограммам. Это упроще-

ние применимо к условным операторам, вложенным в **else**-часть других условных операторов:

```
if condition1 then                                                    [9]
    …
else
    if condition2 then
    …
    else
        if condition3 then
        …
        else
            …
            …Последующие вложенные вхождения if … then … else … end …
            …
        end
    end
end
```

В такой структуре вложенность производит обманчивое впечатление сложности, в то время как фактическая структура решения — последовательная:

- если имеет место *condition*1, выполните первую **then**-часть и ничего более;
- для $i > 1$, если имеет место $condition_i$, но ни одно из предыдущих условий $condition_j$ не выполнено для $j < i$, выполните i-ю **then**-часть и ничего более;
- если ни одно из условий $condition_i$ не выполняется, то выполните наиболее вложенную **else**-часть и ничего более.

Ключевое слово **elseif** позволяет удалить излишнюю вложенность в этом случае, написав последовательные варианты на одном уровне:

```
if condition1 then                                                    [10]
    …
elseif condition2 then
…
elseif condition3 then
    …
elseif … Последующие условия при необходимости then
    …
else – Как ранее, else-часть является возможной
    …
end
```

Первый вариант структуры подобен матрешке:

Второй вариант гребенчатой структуры не столь хитроумен, но более понятен:

Ключевое слово **elseif**, написанное как единое слово, не следует смешивать с парой подряд идущих слов **else** и **if**, использованных в варианте с матрешкой, поскольку в этом случае каждое **if** должно иметь свои собственные **then** и **end.**

Используя **elseif**, мы можем переписать пример **[8]** в виде одного условного оператора без применения вложенности:

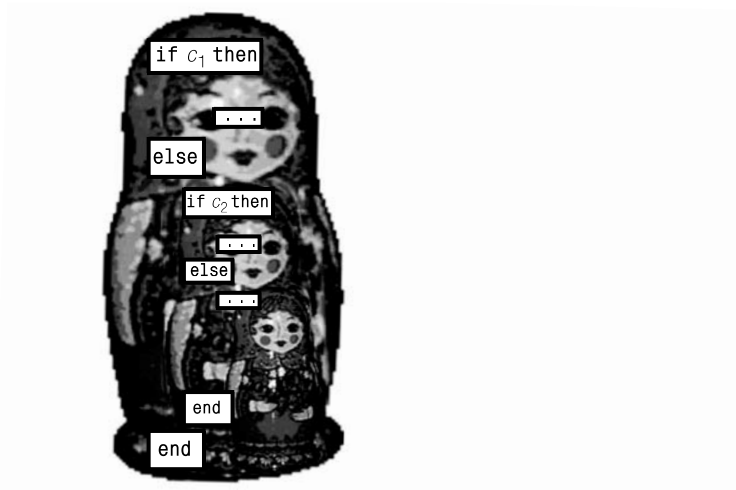


Рис. 7.11. Матрешка

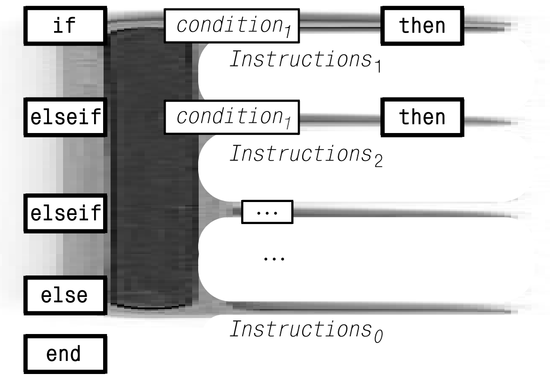


Рис. 7.12. Гребенчатая структура

```
from … invariant … until … loop                                    [11]
              – Опущенные предложения цикла такие же, как в [7]
   if Line8 .item .is_exchange then
      show_blinking_spot (Line8 .item .location)
   elseif Line8 .item .is_railway_connection then
      show_spot (Line8 .item .location)
   else
      show_spot (Line8 .item .location)
   end
   Line8 .forth
variant … end
```

Условный оператор: синтаксис

Приведем сводку форм условного оператора.

Синтаксис

Условный оператор
Условный оператор состоит в порядке следования из:
- «If-части» в форме **if** *условие*;
- «Then-части» в форме **then** *составной оператор*;
- нуля или более «Else-if» частей, каждая в форме **elseif** *условие* **then** *составной оператор*;
- нуля или более «Else»-частей, каждая в форме **else** *составной оператор*;
- ключевого слова **end**.

Каждое *условие* является булевским выражением.

Кстати, если вы находите, что этот способ описания синтаксиса многословен и недостаточно точен, то вы правы. Лучший способ описания таких конструкций дает нотация, известная как нотация БНФ. Мы познакомимся с ней в главе, посвященной синтаксису. Неформальные спецификации, сопровождаемые примерами, пока достаточны для понимания.

Условный оператор: семантика

Предыдущее обсуждение позволяет понять эффект действия условного оператора.

Семантика

Условный оператор
Выполнение условного оператора состоит в выполнении не более одного составного оператора, входящего в одну из частей: «Then», «Else If», если присутствует, «Else», если присутствует. Какой оператор будет выполняться?
- Если условие, следующее за **if**, имеет значение **True**, то выполняется составной оператор части Then.
- Если это условие ложно и существуют Else_if, то первая из таких частей, для которой выполняется условие, и определяет выполняемый составной оператор.
- Если ни одно из рассмотренных выше условий не выполняется и существует Else-часть, то она определяет выполняемый составной оператор.
- Если ничего из выше рассмотренного не применимо, то никакой составной оператор не выполняется и действие условного оператора в этом случае эквивалентно пустому оператору.

Условный оператор: корректность

Корректность условного оператора определяется корректностью каждой его ветви в предположении, что выполняется условие для этой ветви.

Корректность

Составной оператор
Чтобы условный оператор **if** *c* **then** *a* **else** *b* end был корректным, следует убедиться, что перед его выполнением:

- если *c* имеет место, то должно выполняться предусловие *a*;
- если *c* не выполняется, то должно выполняться предусловие *b*.

Из конъюнкций постусловий *a* and *b* – и условия, при котором каждое из них выполняется, – должно следовать желаемое постусловие для всего составного оператора.

Обобщите самостоятельно это правило на общую конструкцию, включающую предложения **elseif**.

7.7. Низкий уровень: операторы перехода

Комбинация наших трех фундаментальных механизмов – последовательности, цикла и условного оператора – дает необходимую основу (будучи дополненной подпрограммами) для построения управляющих структур, необходимых для написания программ.

Эти механизмы языка программирования имеют двойников в машинном коде, который для большинства компьютерных архитектур имеет много рудиментов. Обычно нам не приходится встречаться с машинным кодом, поскольку компиляторы ответственны за перевод программного текста с одного уровня на другой. Однако полезно понимать, как механизмы, доступные на уровне аппаратуры, способны управлять потоком выполнения нашей программы.

Условное и безусловное ветвление (переходы)

Управляющие механизмы машинных языков (язык ассемблера, система команд компьютера) обычно включают:

- **Безусловный переход**: команду, которая передает управление команде с указанным адресом расположения в памяти. В ниже приведенном примере такая команда появляется как **BR** *Address*, где *Address* – адрес памяти целевой команды;
- **Условный переход:** передает управление по заданному адресу при выполнении условия или следующей команде, если условие не выполняется. В нашем примере проверяется условие равенства двух значений: **BEQ** *Value1 Value2 Address*. Мнемоника команды – «**B**ranch если **EQ**ual».

Для языка ассемблер (примеры приводятся на этом уровне) адреса являются условными. Фактические адреса памяти появляются в процессе загрузки программы.

Понятие команды с указанным адресом расположения в памяти следует из принципа хранимой в памяти программы, справедливого для всех компьютеров, которые хранят в своей памяти и данные, и программы. Команда перехода, которая ошибочно задает адрес памяти, не являющийся командой, служит причиной ошибочной работы, приводящей, как правило, к завершению работы программы.

Рассмотрим составной оператор:

```
if a = b then
    Составной_1
else
    Составной_2
end
```

В машинном коде это может выглядеть так:

```
100    BEQ loc_a loc_b 104
101    … Код для Составной_2
102    …
103    BR 106
104    … Код для Составной_1 …
105    …
106    … Код для продолжения программы …
```

Здесь *loc_a* и *loc_b* задают адреса памяти, хранящие значения. Числа слева задают адреса команд, начиная с адреса 100 (просто в качестве примера). Определение точного расположения в памяти машинного кода, связанного с каждым программным элементом, является непростой задачей. Ее решением занимаются разработчики компиляторов, а не прикладные программисты, специализирующиеся на разработке приложений.

По примеру кода для *условного оператора* вам придется, выполняя упражнение, построить структуру кода, который компилятор строит для *цикла*.

Оператор goto

Команды перехода, условные и безусловные, отражают базисные операции, которые может выполнять компьютер: проверять выполнение некоторых условий, таких как равенство двух значений, хранящихся в памяти, передавать управление команде, хранящейся в памяти по определенному адресу. Все языки программирования имели в свое время, а некоторые и до сих пор предлагают оператор **goto**, чье имя образовано слиянием двух английских слов «*go to*» (*перейти к*). В таких языках можно приписать каждому оператору программы **метку**:

```
some_label: some_instruction
```

Здесь some_label – это имя, которое вы даете оператору. Общепринято метку отделять от оператора символом двоеточие, хотя возможны и другие соглашения. При компиляции компилятор отображает метки в условные адреса (100. 101, ...), появляющиеся в нашем примере машинного кода. Языки с **goto** включают оператор безусловного перехода в форме:

```
goto label
```

Эффект выполнения этого оператора состоит в том, что управление передается оператору с меткой, заданной оператором перехода.

Вместо красивого условного оператора **if** *condition* **then** *Compound_1* **else** *Compound_2* **end** старые языки программирования имели более примитивный оператор выбора **test** *condition simple_instruction*, выполняющий *simple_instruction*, если *condition* истинно, в противном случае выполнялся оператор, следующий за тестом.

Такой оператор легко отображается в команды машинных языков, такие как **BEQ**. При использовании *goto* эквивалентным представлением оператора test будет следующий код:

```
       test condition goto else_part
       Compound_2
       goto continue
```

```
else_part: Compound_1
continue:… Продолжение программы…
```

Это менее ясно, чем условный оператор с его симметричной иерархической структурой, в особенности с учетом вложенности. Игнорируя часть **from**, цикл можно было бы представить как:

```
start:     test exit_condition goto continue
              Body
              goto start
continue:… Продолжение программы …
```

Поток управления включает два оператора перехода – две ветви, идущие в разных направлениях.

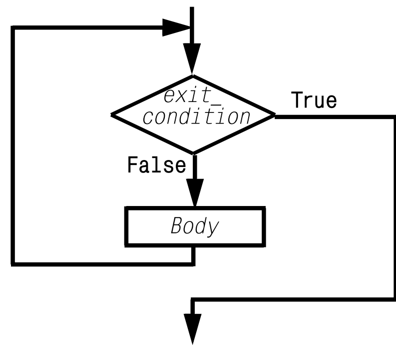


Рис. 7.13. Блок-схема цикла

Блок-схемы

На последнем рисунке поток управления показан в виде так называемой блок-схемы программы или просто блок-схемы. Форма элементов блок-схемы стандартизована: ромбы для условий с двумя выходящими ветвями для True и False; прямоугольник для обрабатывающих блоков, здесь для *Body*. Можно проверить свое понимание блок-схем, изобразив схему для условного оператора.

В свое время блок-схемы были весьма популярны для выражения структуры управления программы. И сегодня их можно встретить в описании процессов, не связанных с программированием. В программировании они потеряли репутацию (некоторые авторы называют их теперь не «flowchart», а «flaw chart» – порочными схемами). Причины этого понятны. Если язык программирования предоставляет вам безусловный оператор перехода **goto** и условный оператор ветвления, такой как **test** *condition* **goto** *label*, то блок-схемы представляют способ отображения потока управления в период выполнения более понятный, чем программный текст, использующий **goto**. Теперь же такое представление устарело по двум причинам.

- Наши программы делают все более сложные вещи. Большие блок-схемы с вложенностью внутри циклов и условных операторов быстро становятся запутанными.
- Механизмы этой главы – составной оператор, цикл, условный – обеспечивают высокий уровень выражения структур управления. Аккуратно отформатированный про-

граммный текст с отступами, отражающими уровень вложенности, дает лучшее представление о порядке выполнения операторов.

Переход от блок-схем к тщательно отобранным структурам управления противоречит клише «рисунок лучше тысячи слов». В ПО нам необходимы многие тысячи, фактически – миллионы слов, но критически важно, чтобы это были «*правильные*» слова. Из-за проблем с точностью картинки теряют свою привлекательность.

> Корректность программ может зависеть от маленьких деталей, таких как использование условия $i <= n$ вместо $i < n$; лучшие рисунки в мире полностью бесполезны, когда приходится иметь дело с правильным отображением таких аспектов.

7.8. Исключение GOTO и структурное программирование

Блок-схемы – не единственное, что вышло из употребления в программной инженерии с превращением ее в более солидную дисциплину: операторы **goto** также потеряли свою былую славу.

Goto вредны?

Причины потери доверия к goto практически те же, что и для блок-схем. Механизмы, которые мы уже изучили, предлагают лучшее управление. Приведу два аргумента.

- Конструкции цикла и условного оператора лучше читаются, чем **goto** – особенно для сложных структур с большой степенью вложенности. Особых доказательств не требуется – достаточно визуально сравнить оригинальные структуры и их **goto-**аналоги.
- Однако это еще не вся история. Остановившись на трех перечисленных механизмах, мы ограничиваем себя в сравнении с программистом, который может использовать произвольно оператор **goto** или, что эквивалентно, произвольные блок-схемы со стрелками, выходящими из любого блока и входящими в любой блок. Прозвищем таких перепле-

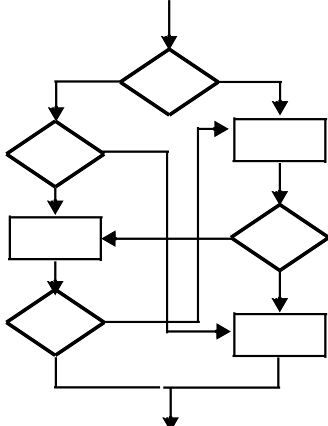


Рис. 7.14. Блюдо спагетти

тающихся структур является «*блюдо спагетти*». Образец такого «блюда» для довольно простого варианта с небольшим числом блоков показан на рисунке. Очевидно, что «наши» структуры управления понятнее и лучше читаются, чем блюдо спагетти. Но, с другой стороны, это ведь только методологический аргумент. Возможно ли, что, ограничив себя тремя структурами и исключив **goto**, мы потеряли нечто важное? Другими словами, существуют ли алгоритмы, не выразимые без полной мощи **goto**?

На этот вопрос ответ получен – **нет**! Теорема, доказанная в 1966 году двумя итальянскими учеными Коррадо Бёмом и Джузеппе Джакопини, говорит, что каждая блок-схема в теории вычислений имеет эквивалентное представление, использующее только последовательности и циклы (нет необходимости даже в условных операторах).

Corrado Böhm, Giuseppe Jacopini: Flow diagrams, Turing machines and languages with only two formation rules. Comm. of the ACM, vol. 9, no. 5, pages 366-371, May 1966.

Общие правила преобразования произвольных блок-схем в программы без goto, приведенные в этой статье, могут быть сложными. Для случаев, встречающихся на практике, часто возможно исключить эти схемы неформальным, но простым и понятным способом. Поскольку это более специальная тема (и она использует присваивание, формально еще не пройденное), ее обсуждение и специальные примеры приведены в приложении. В одном из упражнений, которое предстоит выполнить после чтения приложения, предлагается построить программу без goto для еще одного примера.

Жизнь без goto

Задача исключения, рассматриваемая в приложении, – это не та задача, которую придется решать в повседневной жизни. Нет необходимости писать программу с **goto**, а потом последовательно исключать этот оператор из программы. Следует ясным способом строить нашу программу, непосредственно применяя высокоуровневые структуры управления, которые доказали свою адекватность при построении алгоритмов, простых и сложных.

Почувствуй историю:

Отмена goto

Сегодня «Go to» – почти бранное слово в программировании. Но так было не всегда. В свое время операторы перехода входили в состав основных структур. И вот в марте 1968 года в журнале *Communications of the ACM* была опубликована статья Эдсгера Дейкстры «О вреде оператора Goto». Чтобы избежать задержки с ее публикацией, Никлас Вирт, бывший тогда редактором журнала, напечатал ее в виде «Письма к редактору». Тщательно проведя обоснование, Дейкстра показал, что неограниченные операторы перехода пагубно влияют на качество программ.

Эта статья вызвала массу полемики. Тогда, как и теперь, программистам не нравится, когда их привычки ставятся под сомнение, – что все же временами случается. Но никто так и не смог привести веских аргументов в пользу неограниченного применения goto.

В короткой статье Дейкстры, которую следует прочитать каждому программисту, объяснялись те вызовы, с которыми приходится сталкиваться при проектировании программ.

Рис. 7.15. Дейкстра

Почувствуй мастера:

Дейкстра о программах и их выполнении

Наши интеллектуальные способности довольно хорошо приспособлены для управления статическими отношениями, но наши способности визуализации процессов, протекающих во времени, относительно слабы. По этой причине нам следует (как мудрым программистам, осознающим наши ограничения) сделать все возможное, чтобы сократить концептуальный разрыв между статической программой и динамическим процессом, установить столь простое соответствие между программой (разворачивающейся в пространстве текста) и процессом (разворачивающимся во времени), насколько это возможно.

Edsger W. Dijkstra, 1968

Никто, тогда или позже, не выразил эту мысль лучше. Программа, даже простая, представляет статический взгляд на широкую область возможных динамических вычислений, определенных на широкой области возможных входов. Эта область настолько широка, а во многих случаях потенциально бесконечна, что ее и изобразить невозможно. Тем не менее, чтобы быть уверенными в корректности наших программ, мы должны уметь выводить ее динамические свойства из статического текста.

Структурное программирование

Революция во взглядах на программирование, начатая Дейкстрой, привела к движению, известному как **структурное программирование**, которое предложило систематический, рациональный подход к конструированию программ. Структурное программирование стало основой всего того, что сделано в методологии программирования, включая и объектное программирование

Уже в первой книге по этой теме было показано, что структурное программирование – это не просто управляющие структуры и программирование без goto. Принципиальным посылом было рассмотрение программирования как научной дисциплины, базирующейся на строгих математических выводах (Дейкстра пошел дальше, описав программирование как «одну из наиболее трудных ветвей прикладной математики»).

В массовом сознании остались, в первую очередь, две идеи – исключение **goto** и ограничение управляющих структур тремя видами, рассмотренными в этой главе: последовательностью, циклом и альтернативой, часто называемых «*управляющими структурами структурного программирования*».

Главной особенностью этих структур является то, что все они имеют **один вход** и **один выход**.

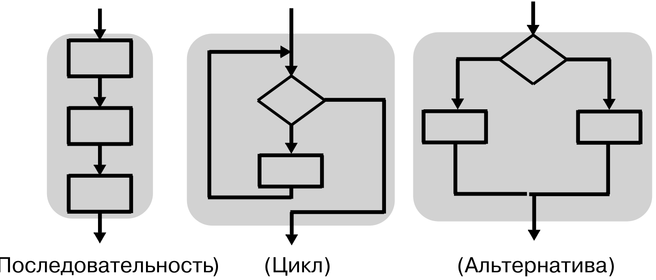


Рис. 7.16. Три вида структур с одним входом и одним выходом

В отличие от этого, произвольные структуры управления (смотри блоки в блюде спагетти) могут иметь произвольное число входов и выходов. Ограничив себя строительными блоками с одним входом и одним выходом, мы получаем возможность построения произвольных алгоритмов любой сложности с помощью трех простых и надежных механизмов.

- **Последовательное соединение**: используйте выход одного элемента как вход к другому, подобно тому, как электрики соединяют выход сопротивления с входом конденсатора.
- **Вложенность**: используйте элемент как один из блоков внутри другого элемента.
- **Функциональная абстракция**: преобразуйте элемент, возможно, с внутренними элементами, в подпрограмму, также характеризуемую одним входом, одним выходом в потоке управления.

Теорема Бёма – Джакопини говорит нам, что мы не потеряли никакой выразительной силы, ограничив себя этими механизмами. Мы получаем существенные преимущества в простоте программы и ее читабельности, а, следовательно, в гарантировании ее корректности, возможности расширения и повторного использования.

Как goto прячется под маской

Хотя мало кто настаивает на возвращение оператора **goto** в прежнем виде, битва за простые структуры управления еще не окончена. В частности, многие языки программирования поддерживают форму цикла с оператором *break*, представляющим оператор выхода из цикла в любой его точке (этот же оператор выхода может применяться и для многовариантного оператора выбора, изучаемого ниже). Вот возможный пример цикла с оператором *break*:

```
from … until условие выхода loop
    Некоторые операторы
    if другое условие then break end
    Другие операторы
end
```

Предупреждение: это только иллюстрация, а не программа на Eiffel.

Если во время выполнения тела цикла станет истинным «*другое условие*», то цикл прекратит свою работу, не выполнив «Другие операторы», не проверив истинности *условия выхода*, не выполняя дальнейших итераций.

Другой конструкцией подобной природы является оператор, применяемый в теле цикла и позволяющий прервать выполнение итерации в любой точке и перейти к выполнению следующей итерации.

Такие операторы – это тот же старый **goto** в овечьей шкуре. Относитесь к ним так же, как и к **goto**.

Почувствуй методологию

Используйте только блоки с одним входом, одним выходом
Избегайте операторов *break* и любых подобных механизмов.

Достаточно просто, применив данные ранее советы, избавиться от скрытого *goto*, переписав пример следующим образом:

```
from … until условие_выхода loop
    Некоторые операторы
    if not другое_условие then
       Другие операторы
    end
end
```

Другие примеры могут потребовать бо́льших усилий на избавление от **goto**, но и они не нарушают общего правила.

Основные возражения против скрытого **goto** остаются те же, что и против общего **goto**, – ясность и простота структур с одним входом и одним выходом. Существует и фундаментальный критерий: наша способность выводить семантику программу из ее текста – в терминах Э. Дейкстры «сократить концептуальный разрыв между статической программой и динамическим процессом». Для циклов, как ранее мы рассмотрели, ключевым приемом построения вывода является **принцип постусловия цикла**: для понимания того, что делает цикл, достаточно скомбинировать инвариант цикла (даже неформальный) с условием выхода. В приводимом ранее примере нахождения максимума множества чисел мы имели инвариант:

“*max* is the maximum of N_1, N_2, …, N_i ”

и условие выхода:

$i = n$

Достаточно просто проверить, что инициализация делает инвариант истинным, что тело сохраняет его истинность и что цикл завершается. Комбинация этих свойств незамедлительно влечет, что *max* является максимумом из *N*1, *N*2, ..., N_n. Если бы мы ввели оператор **break** или другую подобную конструкцию, то такой простой вывод стал бы невозможным, само понятие инварианта цикла перестало бы существовать, по крайней мере, в такой простой форме, как мы его изучали. Это еще одна причина не отказываться от схем с одним входом и одним выходом.

Риск ошибок при использовании подобных конструкций – не просто теоретический. Основная телефонная сеть США AT&T в 1990 году вышла из строя, отключив все теле-

фоны. Причиной была ошибка в программном коде на языке C: оператор break прервал выполнение оператора switch, хотя предполагалось, что он прервет выполнение охватывающей конструкции.

7.9. Вариации базисных структур управления

Последовательность, цикл, альтернатива – триада «структурного программирования» – составляет базис структурного потока управления. Но есть некоторые интересные варианты, заслуживающие отдельного рассмотрения.

Согласно теореме Бёма – Джакопини триады достаточно для выражения всех имеющих смысл алгоритмов, ни одно из расширений не является теоретически необходимым – все они могут быть выражены как комбинация элементов триады. Но это не исключает их практической полезности для программистов, так как они в частных случаях предлагают более эффективный способ записи. Исходя из этого критерия, мы можем разделить их на две категории.

1. Конструкции, обеспечивающие в сравнении с базисными серьезное улучшение для важных частных случаев.
2. Механизмы, которые следует знать, так как они присутствуют в некоторых широко распространенных языках программирования, хотя и нет веских аргументов для их использования.

Разница в конструкциях, во многом, дело вкуса, так что у вас может быть свое собственное мнение.

Инициализация цикла

Предложение **from** в конструкции нашего цикла представляет способ задания начальных характеристик управления. Оно, конечно, избыточно, так как вместо такой конструкции:

```
from
    Операторы инициализации
until условие loop
    Тело
end
```

можно скомбинировать две конструкции – последовательность и цикл, записав:

```
Операторы инициализации
from
    <- Здесь пусто
until условие loop
    Тело
end
```

Такая запись дает тот же самый эффект. Конструкции циклов в некоторых языках программирования действительно не включают инициализацию и начинаются с **until** или его эквивалента.

Причина включения предложения **from** в синтаксическую конструкцию цикла в том, что большинству циклических процессов, подобно процессам аппроксимации, нужна инициа-

лизация, чтобы цикл мог правильно работать. Инициализация – это не просто группа операторов, выполняемых перед началом цикла, – это неотъемлемая часть цикла. Правила корректности цикла отражают этот факт, приписывая инициализации важную роль – о**беспечить начальную истинность инварианта цикла** еще до того, как начнут выполняться итерации, заданные телом цикла, которые должны затем поддерживать инвариантность.

В языках, чьи циклы не имеют предложения **from**, приходится выполнять инициализацию как независимый составной оператор, чаще всего с комментарием, поясняющим его роль.

В Eiffel это обсуждение дает нам ответ на вопрос: если некоторые операции выполняются перед циклом, то они должны появляться в операторах, предшествующих циклу, или в предложении **from**? В зависимости от роли этих операций они могут появляться в том или в другом месте или могут быть расщеплены на две части.

Почувствуй методологию

Где размещать действия, предшествующие циклу

Если оператор, выполняемый перед циклом, служит для инициализации циклического процесса, в частности, для формирования инварианта, то размещайте его в предложении **from**.

Если же часть из множества операций просто по алгоритму следует выполнить до начала цикла, то помещайте их перед циклом как независимые операторы.

Другие формы цикла

Многие языки программирования предлагают форму цикла с ключевым словом, задающим *условие продолжения* работы цикла, а не условие его окончания:

```
while условие_продолжения loop
    Тело
end
```

Семантика понятна: вычисли «*условие_продолжения*»; если оно истинно, то выполни итерацию и снова проверь условие, если условие ложно, то цикл завершает работу. Это эквивалентно записи в нашем стиле:

```
until not условие_продолжения
```

или **until** *условие выхода*, где *условие выхода* является отрицанием *условие_продолжения*.

Разница в следующем.

- Форма **while** отражает динамику: во время выполнения цикл будет повторять тело до тех пор, пока истинно *условие_продолжения*.
- Форма **until** характеризует выводимость – корректность цикла и его эффект: она отражает тот факт, что постусловие цикла и его инвариант определяют *условие_выхода*.

Еще одну форму цикла не следует путать с формой **from**... **until**... **loop** ... **end**, хотя она тоже использует обычно ключевое слово **until**, но в конце конструкции, а не в начале. Ее типичный вид:

```
repeat
    Тело
```

```
until
    условие_выхода
end
```

Семантика такова: выполните тело, если после этого условие выхода истинно, то выйдите из цикла, в противном случае начните все сначала. В отличие от предыдущих вариантов (**from** ... **until** ... и **while**), где тело цикла может ни разу не выполняться, в данном варианте тело цикла всегда будет выполняться, по крайней мере, один раз.

Нетрудно выразить этот вариант в нашей нотации:

```
from
    Тело
until
    условие_выхода
loop
    Тело
end
```

Недостатком является повторение тела цикла, в то время как мы обычно стараемся избежать повторения кода. Можно избежать повторения, превратив тело, включающее несколько операторов, в подпрограмму.

Здесь мы достигли точки, где возможны разные мнения. Одни предпочитают иметь несколько конструкций с возможностью проверки условия выхода (условия продолжения) в начале или в конце цикла в зависимости от возникающей ситуации. Я предпочитаю иметь единственную конструкцию цикла с тщательно определенной семантикой и простым понятием инварианта (чей двойник для цикла в форме **repeat** является более сложным). Я готов заплатить за это в ряде редких случаев повторением строчки кода или добавлением подпрограммы.

Необходимость действительно редкая, так как циклы с повторением «ноль или более раз» встречаются значительно чаще, чем циклы, требующие «одно или более повторений».

Еще одним общим видом является цикл типа **for**:

```
for i: 1 .. 10 loop
    Тело
end
```

Семантика такова: выполни тело, чьи операторы обычно используют *i*, последовательно для всех значений *i* в заданном интервале (здесь 1, 2 и так далее до 10). Граничные значения 1 и 10 даны для примера, и обычно они задаются значениями переменных или выражений, а не константами.

В языке С и его последователях, таких как С++, форма такова:

```
for (i=1; i <= 10; i++) {
    Тело
}
```

Первый элемент в круглых скобках задает инициализацию *i*. Второй – условие продолжения. Последний элемент представляет операцию увеличения, выполняемую после каждого

выполнения **тела**. Нотация *i*++ означает увеличение *i* на единицу. Отметим видимую разницу в стиле синтаксиса: вместо ключевых слов используются символы, такие как скобки, точки с запятой, что характерно для этого стиля программирования.

> Формы цикла, основанные на явном использовании индексов известны также, как циклы **do**, от соответствующего ключевого слова языка Фортран, который ввел подобный механизм, будучи первым языком программирования.

Конструкция **from** этой главы выражает такие циклы следующим образом:

```
from
    i:= 1
until
    i > n
loop
    Тело
    i:= i + 1
end
```

Здесь используется оператор присваивания *a:= b* (дать *a* текущее значение *b*), изучаемый в деталях в главе 9.

На этом история со стилем **for** для цикла не заканчивается. Показанный его эквивалент **from** ... **until** ... **loop** не столь хорошо выполняет работу. Выполняя операции над индексом в разных местах: инициализации, теста, увеличения индекса – он скрывает основную идею итерирования некоторого интервала, 1 ... 10 в нашем примере. Поэтому появляются веские основания для формы цикла более высокого уровня, просто предписывающего: «Выполни эту операцию **для всех элементов** данного множества».

Цикл **for** – пример движения в этом направлении. Здесь данное множество представлено непрерывным интервалом целых чисел. Хотелось бы выполнять итерации на множествах более общего вида, например, на списках, таких как линия метро, представленная списком станций. Общий механизм должен позволять в терминах высокого уровня говорить нечто подобное: «*Выполни эту операцию для всех станций этой линии*».

Такой механизм имеет имя: **итератор**. При обсуждении структур данных мы увидим, что можно, не вводя новые структуры управления, определить мощные итераторы, применимые к широкому спектру структур данных.

Множественный выбор

Условный оператор, как мы видели, решает проблему разделения области задачи на непересекающиеся подмножества, в каждом из которых решение ищется независимо. Базисная форма **if** ... **then** ... **else** использует два подмножества. Включение **elseif** позволяет разбивать область на произвольное число частей.

Выберете ваш язык:

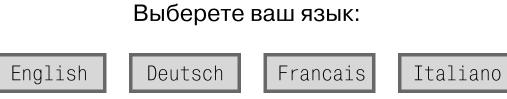


Рис. 7.17. Выбор из списка

Многие языки программирования предлагают для этих целей другую конструкцию выбора, когда речь идет о выборе из некоторого множества чисел или символьных значений. Рассмотрим приложение с графическим интерфейсом, предоставляющее пользователю право выбора языка интерфейса для дальнейшего общения с пользователем.

Предположим, что в приложении введена целочисленная переменная *choice*, принимающая значение от 1 до 4, в зависимости от выбора пользователя:

```
if choice = 1 then                                                    [12]
    … Выбрать английский в качестве языка интерфейса …
elseif choice = 2 then
    …
elseif choice = 4 then
    … Выбрать русский в качестве языка интерфейса …
else
    … Этот вариант не должен встречаться (смотри ниже) …
end
```

В этом случае множественный выбор доступен в более компактной форме:

```
inspect                                                               [13]
    choice
when 1 then
    … Выбрать английский в качестве языка интерфейса …
when 2 then
    …
when 4 then
    … Выбрать русский в качестве языка интерфейса …
else
    …
end
```

Сделанные упрощения являются довольно скромными, но они отражают общую идею: если все условия выбора сводятся к форме $choice = val_i$ для одного и того же выражения *choice* и различных констант val_i, то нет необходимости в повторении «*choice* =». Вместо этого можно просто перечислить значения констант в последовательно идущих предложениях **when**.

Это нотация, используемая в Eiffel. В языках Pascal и Ada есть подобная конструкция с ключевыми словами **case** ... **of**. В языке C и его последователях (C++, Java, C#)) есть оператор **switch**, не в точности соответствующий нашей нотации, но позволяющий получить ее эквивалент.

Множественный выбор доступен только для выражений определенного типа. В Eiffel его можно использовать только для целых и символов. В обоих случаях:

- применяется простая нотация для выбора значений, такая как 1 для целых и _A_ для символов;
- значения упорядочены. Как следствие, можно создавать интервалы: целых, такие как 1 ... 10, символов, такие как ′A′ ...′Z′.

В предложениях **when** можно использовать интервальную нотацию:

```
inspect
    last_character -- Символ, введенный пользователем
when 'a'.. 'z' then
    … Операции для случая ввода буквы латиницы в нижнем регистре …
when 'A' .. 'Z' then
    … Операции для случая ввода буквы латиницы в верхнем регистре …
when '0' .. '9' then
    … Операции для случая ввода цифры …
else
    … Операции для случая, когда ввод не является буквой или цифрой …
end
```

В предложениях **when** можно также перечислять несколько значений или несколько интервалов или их смесь, используя запятые в качестве разделителей:

```
inspect                                                                    [14]
    код заказа   – при заказе авиабилетов
when 'A', 'F', 'P', 'R' then
    … Операции при покупке билетов первого класса …
when 'C' .. 'D', 'I .. 'J', then
    … Операции при покупке билетов бизнес класса …
when 'B', 'H', 'K .. 'N', 'Q', 'S .. 'W', 'Y' then
    … Операции при покупке билетов эконом класса …
else
    … Обработка случая для нестандартного кода заказа …
end
```

С такими возможностями преимущества **when** нотации становятся ощутимыми. Кроме того, многовариантный выбор позволяет задать содержательное правило «**отсутствия пересечений**»: никакие значения не могут появляться в двух различных when-ветвях. Компиляторы следят за этим, выдавая ошибку при обнаружении двусмысленности.

В условном операторе **if** *c1* **then** ... **elseif** *c2* **then** ... **elseif** *c3* **then** ... **end** вполне возможно одновременное выполнение нескольких условий *c1*, *c2*, *c3*, ... Явно заданный последовательный характер проверки определяет, что будет выполняться первая ветвь, для которой выполняется условие. Множественный выбор с **when** требует, чтобы только одно условие было истинным.

> Различные варианты условного оператора все же по умолчанию носят исключающий характер. Как определяет его семантика, *i-я* ветвь соответствует случаю, когда ее условие выполняется и ни одно из условий предыдущих ветвей не выполнилось. Динамически это ведет к непересекающимся случаям. Во множественном выборе отсутствие пересечений задано при проектировании.

Семантическое различие открывает путь к более эффективной реализации множественного выбора. Поскольку условия не должны вычисляться последовательно и благодаря принципу хранимой программы, применима техника, известная как **таблица переходов.** Строго говоря, она применяется разработчиками компиляторов, а прикладные программисты непосредственно ее не используют, но полезно знать саму идею.

Предположим, необходимо реализовать множественный выбор для случая, когда выбор определяется значениями целых чисел, как в нашем первом примере с выбором языка интерфейса; для общности положим, что выбор определяется значениями *n* чисел. В условном операторе пришлось бы последовательно проверять, равна ли переменная *choice* 1, если нет, то равна ли она 2, и так далее, выполняя *n* сравнений в худшем случае.

Используя дополнительные знания о том, что выбор осуществляется между непересекающимися значениями целочисленного интервала и что программы для каждого случая хранятся в памяти по определенным адресам, можно ввести специальную структуру данных – таблицу переходов, позволяющую сразу же найти нужный элемент для любого из вариантов.

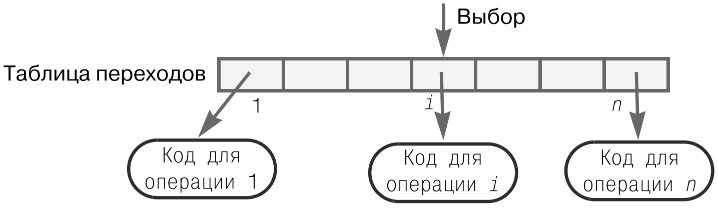


Рис. 7.18. Использование таблицы переходов для организации множественного выбора

Таблица переходов соответствует массиву – структуре данных высокого уровня, с которой мы вскоре познакомимся. Каждый из ее элементов содержит адрес кода, выполняемого для соответствующей ветви **inspect**. На рисунке это показано в виде стрелки, указывающей на точку хранения кода. Двойник «стрелки» есть в языках высокого уровня: понятие ссылки или указателя. Трансляция множественного выбора в машинный код проста: выражение *choice* используется для нахождения нужного элемента в таблице переходов, этот элемент содержит адрес, выполняется код, расположенный по этому адресу.

Важное свойство такой схемы – она не требует последовательных проверок, только доступ к массиву по индексу и переход по полученному адресу. Обе операции выполняются за константное время вне зависимости от числа ветвей.

Предваряя методы, с которыми мы столкнемся при изучении структур данных, мы видим, что этот подход включает *компромисс память-время*: в стремлении получить выигрыш во *времени* исполнения мы жертвуем некоторой *памятью* для хранения таблицы переходов.

Этот пример показывает великолепие техники таблицы переходов, когда все значения принадлежат некоторому интервалу. Для более сложных случаев, как в примере с заказом авиабилетов, преимущества по сравнению с последовательной проверкой не кажутся столь убедительными, но в этом и состоит работа создателей компиляторов, чтобы проектировать лучшую комбинацию обоих подходов для каждого частного случая.

Почувствуй историю

Когда методы реализации влияют на проектирование языка

Исторически операторы множественного выбора появились в языках программирования как следствие таблицы переходов, применяемой в реализации. Первый аналог появился еще в языке Фортран в виде оператора, получившего название «Вычисляемый Goto» в форме GOTO (*L*1, ..., *Ln*), *CHOICE*. Если целочисленное выражение *CHOICE* имело значение *i*, то происходил

переход по метке *Li*. Оператор switch языка C и его последователей близок по смыслу. Такие конструкции непосредственно отражают технику таблицы переходов.
Тони Хоар (C.A.R. Hoare) предложил свободную от goto конструкцию **case**, включенную Виртом в спроектированные им языки Algol W и Pascal. Она и является источником современных форм множественного выбора.

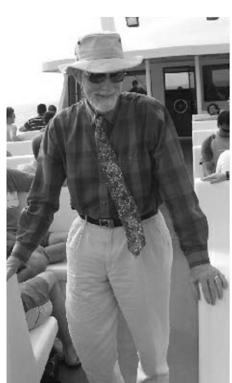


Рис. 7.19. Хоар (2007)

При проектировании множественного выбора необходимо решить, что делать в случае, когда ни один из указанных вариантов не имеет места. Оператор **inspect** может включать предложение, обрабатывающее этот случай. Как и для случая условного оператора, это предложение не является обязательным и может быть опущено. В этом случае оба оператора выполняются по-разному.

- В отсутствие предложения **else**, когда ни одно из условий не выполняется, условный оператор по своему действию эквивалентен пустому оператору.
- Для **inspect** такая ситуация приведет к ошибке периода выполнения и возникновению исключительной ситуации.

Эта политика вначале может вызывать удивление. Причина в том, что множественный выбор явно перечисляет множество возможных вариантов и действия, предпринимаемые для каждого из них. Если возможны и другие значения, то для их обработки и следует указать предложение. Если такое предложение не включается, то это предполагает, что вы ожидаете появление только значений указанного множества. В случае с выбором языка значением может быть только от 1 до 4. Если эти ожидания нарушены, то ничего не делать – это неправильная идея, так как, вероятнее всего, она приведет к некорректным вычислениям и другим проблемам, ведущим к краху. Гораздо лучше захватить источник проблем в начале его появления и включить исключение.

Это обоснование неприменимо к условному оператору, который последовательно проверяет условие за условием, выполняя некоторые операции, когда условие становится истинным.

7.10. Введение в обработку исключений

«Все счастливые семьи похожи друг на друга, каждая несчастливая семья несчастлива по-своему», – все знают эти первые строчки романа «Анна Каренина». Счастливые выполнения

программ все используют одни и те же структуры управления; несчастливые – несчастны по многим различным причинам, называемым *исключениями*.

Исключения дополняют структуры управления этой главы, обеспечивая способ управления специальными случаями, не повреждая нормальный поток управления. Поскольку мы не будем нуждаться в исключениях для структур данных и примеров алгоритмов в этой книге, – фактически, они зарезервированы на непредвиденные случаи, – в этом разделе вводятся только основные идеи.

Вы можете рассматривать этот материал как дополнительный и пропустить его (вместе с последующим разделом) при первом чтении.

Роль исключений

Что представляет «специальный» или «исключительный» случай? Мы увидим, что это понятие можно определить вполне строго, но пока будем полагаться на интуицию, считая, что это событие, которое не должно было случиться. Вот примеры некоторых событий, которые могут, каждое по-своему, разрушить блаженство нормального выполнения программы.

- Деление на ноль, или другие арифметические неприятности.
- Попытка создать объект после превышения пределов памяти.
- Вызов void, который мы определили как попытку вызвать метод несуществующим объектом (*x.f*, где *x* имеет значение void).
- Нарушение контрактов, если вы включили механизм мониторинга предусловий, постусловий и инвариантов в период выполнения.
- Выполнение оператора, сигнализирующего о возникновении ошибочной ситуации.

Такие события включают исключения. Во всех случаях, за исключением последнего, это будут *системные* исключения, возникающие по внешним причинам. В последнем случае сама программа явно вызывает исключение, которое относится к *программистом определенным* исключениям. Это различие иллюстрирует две различные роли обработки исключения.

- Можно использовать обработку исключения как **способ последней надежды** обработать неожиданное событие, из-за которого нормальный поток управления потерпел неудачу. Нереально защищать каждое деление тестом: а не является ли знаменатель равным нулю? Нереально при каждом создании объекта проверять, а есть ли достаточно памяти. Еще труднее планировать некоторые другие ситуации – отказы аппаратуры, прерывания, инициируемые пользователем. Исключение позволяет программе восстановиться или, по крайней мере, с достоинством закончить свою работу, когда любой из ее операторов был прерван из-за возникновения неожиданного события. «Неожиданное» в данном контексте означает событие, не предусматриваемое при работе этого оператора.
- Случай исключений, определенных программистом, отличен: здесь обработка исключений становится **управляющей структурой**, которую следует добавить к нашему каталогу.

Точные рамки для обсуждения отказов и исключений

Для проектирования подходящей стратегии использования исключений – в частности, определения того, что является «нормальным», а что «ненормальным», «исключительным», – будем основываться на понятии Проектирования по Контракту, определенном в предыдущих обсуждениях. Мы знаем, что каждая подпрограмма, независимо от ее конкретной реализации, позволяет задать:

- предусловие: требования к вызывающей программе, которые должны быть выполнены непосредственно перед вызовом;
- постусловие: истинное в конце выполнения и отвечающее интересам вызывающей программы;
- инварианты охватывающего класса: общие свойства, необходимые всем методам класса.

Этот подход можно расширить и на операции, не являющиеся подпрограммами, например, на сложение с плавающей точкой, на создание объектов и так далее.

В идеальном мире все семьи были бы счастливы, и все операции соответствовали бы своим контрактам. В реальном мире операции по тем или иным причинам не всегда выполняют свои обязательства. Если есть распределитель памяти, но вся доступная память уже распределена, то он не сможет создать объект. Если есть подпрограмма, некорректно запрограммированная, то она не сможет выполнить постусловие. Этих наблюдений достаточно для выработки точных определений.

Определения: Отказ, Исключение, Получатель

Отказ – неспособность операции выполнить свой контракт.
Исключение – отказ одной из операций во время выполнения подпрограммы.
Получатель – подпрограмма, в которой возникло исключение.

«Отказ» является первичным понятием и применяется к операциям любого вида: подпрограммам, но также и к базисным операциям, таким как создание объекта. «Исключение» является производным понятием и применимо только к подпрограммам: в подпрограмме возникает исключение, если она выполняет операцию (базисную или вызов подпрограммы), приводящую к отказу.

Почему исключение – не то же самое, что и отказ? Часто появление исключения приводит к отказу у получателя, но не всегда! Подпрограмма может предусмотреть ***обработку исключения*** – *код спасения*, который попытается исправить положение, повторно запустит программу с лучшим исходом.

Если же подпрограмма не предусмотрела обработку исключений или не смогла исправить положение, то выполнение программы заканчивается отказом. В этом случае она включает исключение у вызывающей программы, у которой снова есть две возможности: отказ или восстановление. В результате исключения распространяются вверх по цепочке вызовов. Если ни одна из подпрограмм в этом процессе не восстановит нормальное выполнение, то программа «завершится с ошибкой».

Мы уже сталкивались с примером исключения, причиной которого был «void-вызов».

Теперь мы знаем, что означает отказ для подпрограмм: быть получателем исключения, не способным восстановить ситуацию.

Повторение

После возникновения исключения можно ли в процессе его обработки все восстановить? Можно, например, применить альтернативную стратегию, когда исходная стратегия потерпела неудачу.

> Иногда полезно повторно применить ту же самую стратегию (пренебрегая обвинениями в безумности, приписываемыми Эйнштейну, – «безумно повторять ту же самую вещь и ожидать других результатов»). Безумно или нет, этот метод может работать, если причиной исключений, например, являются так называемые самоустраняемые сбои аппаратуры.

Для иллюстрации идеи позвольте показать, как работает в Eiffel механизм исключений, базирующийся на двух ключевых словах – **rescue** и **Retry** («спаси» и «повтори»). Предложение **rescue**, введенное в подпрограмму, выполняется, когда программа становится получателем исключения. **Retry** является предопределенной булевской переменной, которая, если она принимает значение истина в конце **rescue**, становится причиной того, что тело выполнится снова; если же **Retry** ложно, то выполнение подпрограммы приводит к отказу. Вот пример:

```
transmit (m: MESSAGE)
        -- Передать m, если возможно.
    local
        i: INTEGER
    do
        send_to_transmitter (m, i)
    rescue
        i:= i + 1
        Retry:= (i <= count)
    end
```

Предполагается, что вызов *send_to_transmitter* (*m*, *i*) пытается послать *m*, используя *i*-й передатчик. Мы располагаем передатчиками по нашему выбору; те, у кого меньшие номера, работают быстрее, но имеют большую вероятность отказа. По этой причине мы вначале пытаемся использовать быстрые передатчики, но в случае их отказа переходим к более надежным.

Подобно любым другим локальным переменным типа *INTEGER,* переменная *i* инициализируется нулем на входе процедуры. Если встретится исключение, как результат отказа вызова *send_to_transmitter*, начнет выполняться предложение **rescue**, увеличив *i* на единицу. Если есть доступные передатчики, **Retry** получит значение true, что приведет к повторному выполнению тела (предложения **do**), передавая сообщение новым передатчиком, который может успешно работать. Если же в **rescue Retry** станет ложным, то все закончится отказом, исключение будет передано вверх по цепочке вызовов.

Этот механизм исключения явным образом разделяет две роли.

E1 Нормальное тело подпрограммы – предложение **do** – не занимается непосредственно обработкой исключения, его задачей является выполнение контракта.

E2 Обработчик исключения – предложение **rescue** – не пытается выполнить контракт, его задача обработать исключение. Подобно сотруднику спасательной службы, он расчищает завалы и создает условия для нормального продолжения программы, если это возможно.

Отказ в подпрограмме сигнализирует, что она не способна выполнить свой контракт. Если это не корневая процедура (верхний уровень выполнения), то это еще не означает, что все выполнение дает отказ, так как подпрограмма передает исключение вверх по цепочке вызовов, и в цепочке может найтись программа, исправляющая ситуацию, благодаря механизму **Retry**, и восстанавливающая нормальный процесс выполнения всей программы.

Как следствие, выполнение может откатиться назад, а потом снова вернуться к вызову *x.r* (...), бывшему причиной отказа, к объекту, присоединенному к *x*. Новый вызов на этом объекте может корректно работать, только если объект находится в согласованном состоянии – состоянии, удовлетворяющем инварианту класса. Правило, сопровождающее исключение, говорит, что в предложении **rescue** в случае, когда **Retry** становится ложным, предваритель-

но следует восстановить инвариант класса. Это и означает «чистку завалов» в случае E2. Это правило отражено в следующем принципе.

Почувствуй методологию

Принцип отказа

Исключение подпрограммы, приведшее к отказу, должно восстановить инвариант класса, а затем стать причиной исключения у вызвавшей подпрограммы.

Принцип позволяет определить, что случится, если получатель исключения не включил **rescue**-предложение. Это случай, характерный для большинства подпрограмм. Обычно программа включает небольшое число предложений **rescue** в наиболее важных критических точках, позволяющих либо восстановить нормальное выполнение, либо достойно завершиться в случае непредвиденных событий. Если исключение возникло в подпрограмме без **rescue**, то эта подпрограмма приведет к отказу, передавая исключение вызывающей подпрограмме. Это соответствует случаю, когда подпрограмма имеет предложение **rescue** в следующей форме:

```
rescue
    default_rescue
```

Здесь *default_rescue* является библиотечной подпрограммой, которая по умолчанию ничего не делает. Хорошая практика состоит в том, чтобы для каждого класса подготовить новую версию *default_rescue* с простой реализацией, восстанавливающей инвариант класса любым доступным образом.

default_rescue наследуется от класса верхнего уровня *ANY*, так что любой класс может переопределить эту реализацию. Эти концепции будут частью изучения наследования.

Детали исключения

В примере с передатчиком мог появляться только один вид исключения. Иногда может потребоваться рассматривать разные виды исключений, предусматривая для них различные способы обработки. Все механизмы обработки исключений позволяют получить доступ к деталям последнего исключения, таким как точный тип исключения. В ОО-языках, таких как Eiffel, Java, C# и C++, эта информация доступна через объект *exception*, создаваемый автоматически, когда появляется исключение.

В Eiffel этот объект доступен через запрос *last_exception*: его тип является потомком библиотечного класса *EXCEPTION*. Исключения, тип которых определен программистом, создаются явно, – для этого используется специальный оператор *raise*; говорят, что он «выбрасывает» исключение, создавая соответствующий объект.

Стиль try-catch обработки исключений

Вместо стиля «rescue-retry», основанного на Принципе Проектирования по Контракту, C++ использует стиль «try-catch», который воспроизведен и в языках Java и C# с небольшими вариациями. Основная идея (детали можно увидеть в приложениях, посвященных этим языкам программирования) состоит в том, чтобы писать код, в котором предусматривается воз-

можность появления исключений, в охраняемых **try** операторах и обрабатывать любые возникшие исключения в одном из его **catch**-предложений.

```
try
      … Обработка общего случая, возможно, создающего исключение …
catch (e: T1, T2)
      … Обработка возникшего исключения типа T1 или T2 …
catch (e: T3, T4, T5)
      … Обработка возникшего исключения типа T3, T4 или T5 …
…
end
```

Для каждого ожидаемого типа предусматривается свой процесс обработки. Оператор **throw** является аналогом *raise*, позволяя программно создавать исключение любого типа. В отличие от **rescue,** предложение **catch** не только «расчищает завалы», но и полностью обрабатывает ситуацию. Этот механизм не поддерживает в явной форме возможность возврата – **retry**, но возврат можно смоделировать, заключив **try** блок в цикл.

Две точки зрения на исключения

Как механизмы языка, оба стиля «try-catch» и «rescue-retry» эквивалентны: все, что выразимо в одном стиле, может быть выражено и в другом стиле. Однако их дух отражает две различные точки зрения на исключения. Согласно одной точке зрения, исключения представляют собой еще одну структуру управления, некий аналог «обобщенного goto», который обрабатывает случаи, отличающиеся от основного, общего случая. Согласно другой точке зрения, исключения – это только адреса вариантов, в частности, системных ошибок, но не способы их обработки. Возможны решения, находящиеся между этими граничными точками зрения, так что при написании сложного ПО вы сумеете выработать подходящий для вас стиль.

7.11. Приложение: Пример удаления GOTO

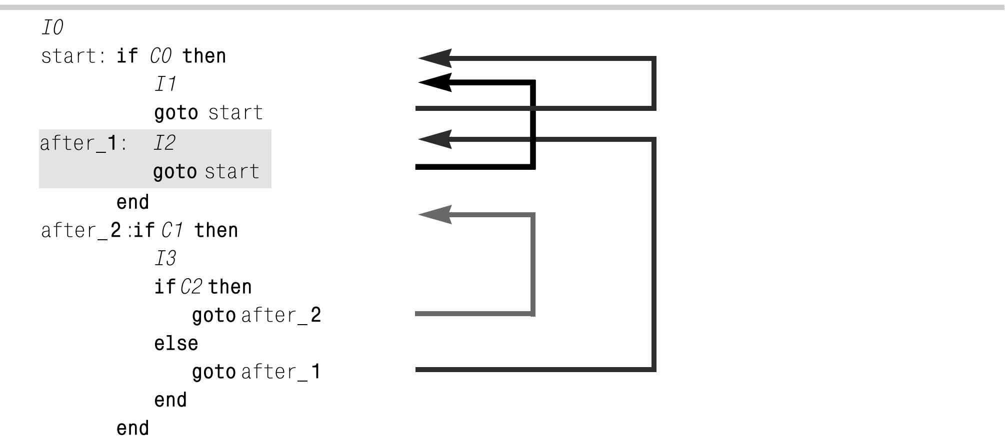
Этот последний раздел дает возможность попрактиковаться в удалении **goto** на специальном примере, довольно простом, но не тривиальном. Это дополнительный материал, который можно пропустить при первом чтении, хотя раздел (также дополнительный) в главе по рекурсии основывается на нем.

Он использует концепции переменной и присваивания, которые формально мы еще не проходили, так что, если это ваша первая книга по программированию, вам следует вернуться сюда после чтения соответствующей главы.

> Если же вы не любите ждать, то вот что нужно понимать: переменная является программной сущностью, которая может принимать различные значения во время выполнения. Атрибуты класса являются переменными, но программа использует и локальные переменные, имеющие смысл только во время выполнения. *Оператор присваивания x:=* *e* позволяет дать переменной *x* новое значение, заданное выражением *e*.

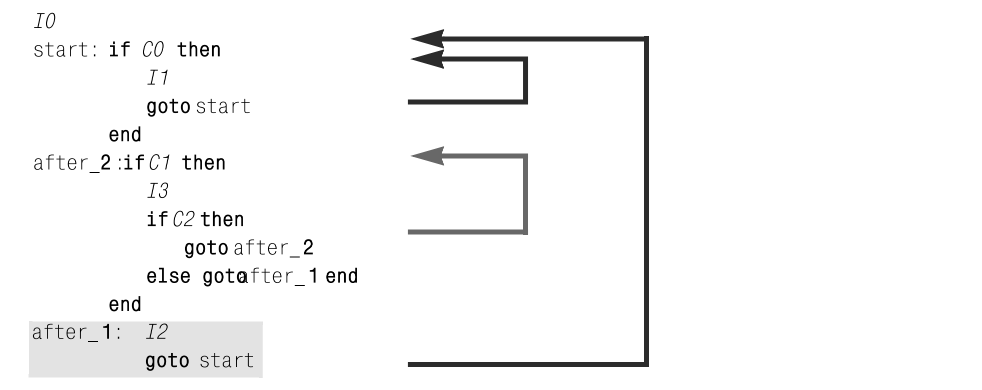
Наш пример по удалению **goto** является не искусственным примером, а схемой, с которой мы встретимся при обсуждении задачи «Ханойская башня». Он включает совокупность опера-

торов **if** … **then** …**else** … **end**, которые могли бы быть представлены в терминах **test** … **goto** … операторов. Для настоящего обсуждения не имеет значения смысл базисных операций – операторов *I0*, *I1*, *I2*, *I3* и условий *C0*, *C1*, *C2*, – достаточно знать, что сами они не содержат ветвлений.



Стрелки справа показывают структуру управления с пересекающимися циклами, что выглядит достаточно запутанно – еще одно блюдо спагетти, – и трудно представить ее в виде структуры без **goto**.

Заметьте, что эта «перепутанность» возникла из-за порядка следования операторов (идущего от оригинального рекурсивного кода, который будет приведен в более поздних обсуждениях). От «перепутанности» можно легко избавиться, так как блок after_1 достижим только через **goto** (оператор, предшествующий ему, сам представляет **goto**, передавая управление по другому адресу). Мы переместим его в другое место, вне других блоков, например, в самый конец.



Спагетти распутаны! Мы видим три цикла с нормальной вложенностью. Остается еще переход **goto** after_1, но так как эта ветвь не используется другими операторами, то весь after_1 блок можно включить в предложение **else**. Так что нетрудно переписать всю структуру без всяких меток, используя вместо этого два вложенных цикла:

```
from I0 until over loop        -- Прежде позиция "start"
    from until not C0 loop
        I1
    end
    from stop:= not C1 until stop loop  -- Прежде позиция "after_2"
        I3
        stop:= ((not C1) or (not C2))
    end
    over:= ((not C1) and C2)
    if not over then I2 end       -- Прежде позиция "after_1"
end
```

Так как у двух циклов условия выхода не элементарны (второму внутреннему циклу перед первой итерацией необходимо выполнение **not** *C1*, а затем **not** *C1* **and not** *C2*), в программе используются булевские переменные *over* и *stop* для представления этих условий. Пример демонстрирует стандартные приемы исключения **goto**.

7.12. Дальнейшее чтение

George Polya: *How to Solve It*, 2nd edition; Princeton University Press, 1957.

На русском языке: Пойа Д. *Как решать задачу*. Учпедгиз, 1959.

Не обращайте внимания на дату публикации George Polya книга по-прежнему остается бестселлером по этой теме.

Edsger W. Dijkstra: *Goto Statement Considered Harmful*, Letter to the Editor, in *Communications of the ACM*, Vol. 11, No. 3, March 1968, pp. 147-148. Доступна в Интернете: www.acm.org/classics/oct95/.

Известная короткая статья «О вреде оператора goto» ознаменовала революцию в методологии и привела к структурному программированию. Объяснила, почему goto неприемлем для хорошего программирования и, что более важно, высветила процесс конструирования программ, лаконично и эффективно. Несмотря на прошедшие десятилетия, ее следует прочитать.

Ole-Johan Dahl, Edsger W. Dijkstra, C.A.R Hoare: *Structured Programming*, Academic Press, 1972.

На русском языке: У.Дал, Э. Дейкстра, К. Хоор *Структурное программирование. Мир 1975*

Классика. Содержит три монографии, первая из которых, «Заметки по структурному программированию» Э. Дейкстры, наиболее известна, но две другие также интересны. Убедительная работа Хоора «О структурной организации данных» дополняет предыдущую работу. Совместная статья Хоора и Дала «Иерархические структуры программ» представляет презентацию языка Симула 67 и излагает концепции, теперь известные как ОО-программирование. Немногие книги оказали такое влияние на развитие программирования.

C.A.R. Hoare and D.C.S Allison: *Incomputability*, in ACM *Computing Surveys*, vol. 4, no. 3, September 1972, pages 169-178. Доступна по подписке: portal.acm.org/citation.cfm?id=356606.

Краткое, простое, ясное объяснение, почему решение некоторых проблем, таких как проблема остановки, не может быть запрограммировано на компьютере.

7.13. Ключевые концепции, изучаемые в этой главе

- Структуры управления определяют последовательность действий при выполнении программы.
- Структуры управления могут рассматриваться как приемы решения задач, сводя исходную, возможно, сложную задачу к множеству более простых задач.
- Базисными структурами управления являются: *составной оператор*, задающий последовательное выполнение списка действий; *условный оператор*, задающий в зависимости от некоторых условий выполнение одного действия из заданного списка; *цикл*, задающий повторное выполнение действия.
- При использовании структур управления следует руководствоваться соображениями корректности. Цикл характеризуется инвариантом, задающим условие, которое поддерживается на всем протяжении цикла, и вариантом, положительным целочисленным выражением, уменьшающимся на каждой итерации цикла. Инвариант позволяет доказать корректность цикла, вариант – его завершаемость.
- Структуры низкого уровня, такие как **goto**, важны на машинном уровне, но с презрением отвергаются языками высокого уровня. Любая программа, использующая их, имеет эквивалент, выраженный в терминах стандартных структур управления.

Новый словарь

Algorithm	Алгоритм	**Branching instruction**	Оператор перехода (ветвления)
Compound	Составной (оператор)	**Conditional**	Условный (оператор)
Concurrent	Параллельный	**Control structure**	Структура управления
Conditional branching	Условный переход	**Cursor**	Курсор
Flowchart	Блок-схема	**Indirection**	Перенаправление
Iterate	Итерирование	**Iteration of a loop**	Итерация цикла
Jump table	Таблица переходов	**Loop**	Цикл
Loop invariant	Инвариант цикла	**Loop variant**	Вариант цикла
Overspecification	Сверхспецификация (избыточная спецификация)	**Parallel**	Параллельный
		Preserve	Сохранение (истинности инварианта)
Sequence	Последовательность	**Space-time tradeoff**	Компромисс «память – время»
Unconditional branching	Безусловный переход		

7-У. Упражнения

7-У.1. Словарь

Дайте точные определения всем терминам словаря.

7-У.2. Карта концепций

Добавьте новые термины в карту концепций, построенную в предыдущих главах.

7-У.3. Циклы в машинных языках.

Рассмотрите цикл в форме:

```
from
    Compound_1
until
    i = n
loop
    Compound_2
end
```

Напишите код на машинном языке, используя команды **BR** и **BEQ**, рассмотренные при обсуждении переходов.

7-У.4. Блок-схема условного оператора

Следуя соглашениям для блок-схем, введенным для циклов, нарисуйте блок-схему условного оператора **if** *Condition* **then** *Compound_1* **else** *Compound_2* **end**.

7-У.5. Бём – Джакопини на практике

Рассмотрите следующий фрагмент программы с **goto**, применяющей условные **goto-**операторы перехода:

```
              Instruction_1
              test c1 goto t3
t2            Instruction_2
t3            Instruction_3
              test c2 goto t2
              Instruction_4
```

1. Нарисуйте соответствующую блок-схему.
2. Предложите фрагмент программы с тем же эффектом, но без **goto**, в котором используются базисные структуры управления: цикл, составной и условный операторы.

7-У.6. Формы цикла

Рассмотрите варианты базисной формы цикла и покажите, как выразить:

1. **repeat** ... **until** ... через **while** ....
2. **while** ... через **repeat** ... **until**.
3. Базисную форму Eiffel (**from** ... **until** ...) через **while** ....
4. Базисную форму Eiffel (**from** ... **until** ...) через **repeat** ... **until** ....

7-У.7. Эмуляция (моделирование) варианта

Вариант цикла обеспечивает доказательство завершаемости. Предположите, что синтаксис не включает **variant-**предложения, но для него по-прежнему задан инвариант. На примере вычисления максимума множества целых докажите завершаемость цикла, перестроив прежнее доказательство с вариантом (подсказка: введите переменную, сохраняющую предыдущее значение выражения варианта, используйте ее для построения инварианта).

7-У.8. Эмуляция retry в языке с try-catch

Рассмотрите схему для обработки исключений rescue-retry, такую как в примере *transmit* с передатчиком сообщения, которая может стать причиной нескольких выполнений главного алгоритма (от нуля до *count* раз). Покажите, как запрограммировать ее в языке программирования, предлагающего обработку исключений в стиле try-catch.

Можно использовать механизмы любого языка: Java, C# или C++, рассматриваемые в приложениях.

7-У.9. Эмуляция try-catch в языке с rescue-retry

Рассмотрите стиль обработки исключений *try-catch*, кратко описанный в этой главе. Покажите, как эмулировать (моделировать) его или один из вариантов (такой как в Java с предложением finally), используя механизмы rescue-retry языка Eiffel.

Для определения типа последнего исключения используйте *last_exception.type*. Нотация {*T*} обозначает объект, представляющий тип *T*, который может быть типом исключения.

8

Подпрограммы, функциональная абстракция, скрытие информации

Управляющие структуры предыдущей главы – цикл, составной и условный операторы, их варианты – дают нам базисные механизмы планирования порядка выполнения операторов. Если бы они были единственными средствами, то нам пришлось бы задавать поток управления со всеми деталями. Для сложных программ глубина вложенности стала бы главным препятствием на пути понимания программы.

Чтобы держать сложность под контролем, привлечем еще один проверенный временем прием решения задач: **выделение подзадач**. Подзадача – это задача, чье решение может помочь в решении других задач. Если мы умеем решать подзадачу и умеем представить это решение в виде элемента управления, простого или сложного, то можно дать имя этой подзадаче и использовать новый элемент под этим именем. Этот прием известен как *функциональная абстракция*, а соответствующий программный механизм – как *подпрограмма* (*routine*).

8.1. Выводимость «снизу вверх» и «сверху вниз»

Почему полезно выделять подзадачи? Можно привести два дополняющих ответа.

- При решении задачи можно выделить подзадачу, для которой решение уже известно. Тогда мы просто вставляем известное решение в решение большой задачи. Это и характеризует подход «снизу вверх» использования подзадач: используем то, что уже нам известно, для решения новых задач, большего размера. Такой стиль построения решения характерен для физиков и инженеров. Инженер, анализируя электрическую систему, опишет ее модель системой дифференциальных уравнений известного типа, затем использует известные методы решения таких уравнений и выведет свойства своей системы.
- В других случаях мы осознаем, что часть решения нашей задачи представляет собой отдельную задачу, которая, мы надеемся, может быть более простой, чем исходная задача. Этот взгляд может быть полезен, даже если мы не знаем решения созданной подзадачи, поскольку он позволяет нам независимо рассматривать отдельные части большой задачи. Мы можем предположить существование решения подзадачи и использовать его для решения задачи в целом. Получив решение в целом, можно вернуться к подзадачам и разыскивать их решение. Это и определяет подход «сверху вниз»: начинаем работать над общей целью и выделяем подцели, которые должны быть решены независимо. Разработка «сверху вниз» известна также как принцип «*разделяй и властвуй*». Мы уже встречались с этим приемом, когда использовали псевдокод, позволяющий неформальным образом описать части программы, которые позже уточнялись в процессе детализации.

Независимо от способа разработки – «снизу вверх» или «сверху вниз» – использование подзадач является формой абстракции: игнорировать специфику частной ситуации, рассматривая ее как экземпляр общей схемы.

В программировании соответствующая конструкция, задающая решение подзадачи, известна как подпрограмма.

Почувствуй терминологию

Подпрограммы под любым другим именем

У английского термина «*routine*» есть синонимы – «*subprogram*» и «*subroutine*».

Подпрограмма может возвращать результат, и тогда ее называют *функцией*. Подпрограмма, не возвращающая результат, называется *процедурой*. Оба эти термина часто используются для ссылок на подпрограммы любого типа. В языке C, например, применяется единственный термин – «функция».

В добавлении к этой терминологической путанице в ОО-языках добавляют новый термин – «метод» (method), означающий то же, что и подпрограмма, но добавляющий дополнительную сложность из-за совпадения с общим употреблением этого слова. Вот пара примеров: «Нет никакого метода в написанных им методах» или «От его методов можно сойти с ума».[1]

Подпрограммы появляются как в разработке «сверху вниз», так и «снизу вверх». При проектировании «снизу вверх» они поддерживают **повторное использование**: можно получить серьезный выигрыш, если вы или кто-то другой полезную для вас алгоритмическую схему превратили в подпрограмму. При подходе «сверху вниз» можно использовать вызовы подпрограмм, которые представляют хорошо определенные элементы разработки, отложив написание самих подпрограмм. Это подобно написанию псевдокода, но в более структурированном варианте, поскольку на момент вызова необходимо дать точное имя и определить интерфейс вызываемой подпрограммы.

8.2. Подпрограммы как методы (routines as features)

Подпрограмма характеризует алгоритм (операцию), применимый ко всем экземплярам класса. Как таковая, она является одним из двух видов характерных компонентов класса, задавая **метод** класса. Другой характеристикой класса, изучаемой в следующей главе, является *атрибут*.

Будучи методом, подпрограмма имеет:

- **объявление**, которое появляется в тексте класса в разделе **feature** и описывает все свойства подпрограммы. Объявление подпрограммы также называется ее **реализацией**;
- **интерфейс**, который выделяет только подмножество свойств подпрограммы, а именно те, которые интересны клиентам подпрограммы. Интерфейс метода отображается в *контрактном облике* класса.

Мы уже сталкивались со многими подпрограммами, зная их как методы класса. Например:

[1] В Eiffel для составляющих класса используется единый термин «feature», который мы, за неимением лучшего, переводим как «компонент». Калька слова feature (фича) (используемая в профессиональном жаргоне) кажется неприемлемой. Компонент – feature – может быть подпрограммой (routine) или переменной, и тогда в Eiffel он называется атрибутом (attribute). Несмотря на справедливую критику термина «метод», именно он, а не термин «подпрограмма», используется при переводе, когда речь идет о методах (routine) класса.

- наш самый первый метод, *explore* в классе *PREVIEW*, является подпрограммой. Это же верно и для метода из предыдущей главы *traverse*, который в разных вариантах предлагался для реализации;
- при изучении того, как использовать класс, зная его интерфейс, мы рассматривали класс *STATION*, который в разделе **feature** объявлял методы – команду *remove_all_segments* и запрос *i_th* (в этом же разделе у этого класса были заданы атрибуты, такие как *south_end* и *count*).

В случае самостоятельного задания метода необходимо дать полное объявление подпрограммы, для использования метода достаточно знания его интерфейса, например:

```
remove_all_segments
        -- Удалить все станции за исключением конечной, юго-западной.
    ensure
        only_one_left: count = 1
        both_ends_same: south_end = north_end
```

Полный текст подпрограммы можно видеть, изучая класс *STATION*. Теперь мы приступаем к изучению того, как самостоятельно писать объявления методов.

8.3. Инкапсуляция (скрытие) функциональной абстракции

Последний пример изучения условного оператора дает хороший пример определения «функциональной абстракции» в форме подпрограммы. Внешний цикл, появляющийся в методе *traverse* из класса *ROUTES*, записан как

```
from … invariant … variant … until … loop                          [1]
    if Line8.item .is_exchange then
        show_blinking_spot ( Line8.item .location)
    elseif Line8.item.is_railway_connection then
        show_big_red_spot ( Line8.item .location)
    else
        show_spot ( Line8.item .location)
    end
        Line8.forth
end
```

В этой записи нас беспокоит не только повторение кода, но и отсутствие распознавания того факта, что действия применяются к одному и тому же объекту – *Line8.item*. Это свойство становится более ясным, если мы выделим условную структуру в самостоятельный метод. Цикл тогда будет выглядеть так:

```
from … invariant … variant … until … loop                          [2]
        show_station(Line8.item)
        Line8.forth
end
```

Он использует новый метод *show_station*, чье объявление появляется в том же классе *ROUTES*:

```
show_station (s: STATION)
        —Подсветить s в форме, соответствующей ее статусу.
    require
        station_exists: s /= Void
    do
        if s.is_exchange then
            show_blinking_spot (s.location)
        elseif s.is_railway_connection then
            show_big_blue_spot (s.location)
        else
            show_spot (s.location)
        end
    end
```

8.4. Анатомия объявления метода

Объявление *show_station* задает типичную форму метода. Многие из его элементов нам уже хорошо знакомы.

Метод является программным элементом, задающим множество операций, которые должны быть выполнены по требованию других программных элементов, называемых **вызовами** метода. Пока у нас есть один пример вызова *show_station* – цикл [2], в котором есть вызов:

```
show_station (Line8.item)
```

Такой вызов обычно появляется в методе; здесь мы предполагаем, что вызов встроен в метод *traverse* того же класса *ROUTES*. В таких случаях говорят, что метод *traverse* является **вызывающим методом** (**вызывателем – caller**) метода *show_station*.

Если метод класса *C* вызывает метод класса *S*, то это делает класс *C клиентом* класса *S*. В данном случае класс *ROUTES* становится своим собственным клиентом.

Во всей системе (программе) метод может быть целью одного, многих или ни одного вызова, но он всегда имеет единственное появляющееся в классе объявление, которое определяет алгоритм метода. Давайте проанализируем ранее приведенное объявление метода *show_station*. Вот его первая строка:

```
show_station (s: STATION)
```

Она задает имя метода и его **сигнатуру**: список его **формальных аргументов**, если они есть, и их **типы**. Формальные аргументы представляют значения, которыми оперирует метод. Каждый вызыватель в момент вызова передает свои значения через **фактические аргументы**, соответствующие формальным аргументам.

Фактические аргументы являются выражениями, тип которых должен соответствовать формальным аргументам.

Данное ранее определение аргумента соответствует как формальному, так и фактическому аргументу.

Сигнатура метода *show_station* включает один формальный аргумент – *s* типа *STATION*. В примере его вызова [2] фактическим аргументом является *Line8.item*. Тип этого выражения действительно принадлежит *STATION*, так как запрос *item* класса *LINE* возвращает станцию в качестве результата. Если же типы несовместимы, то EiffelStudio выдаст сообщение об ошибке на этапе компиляции системы:

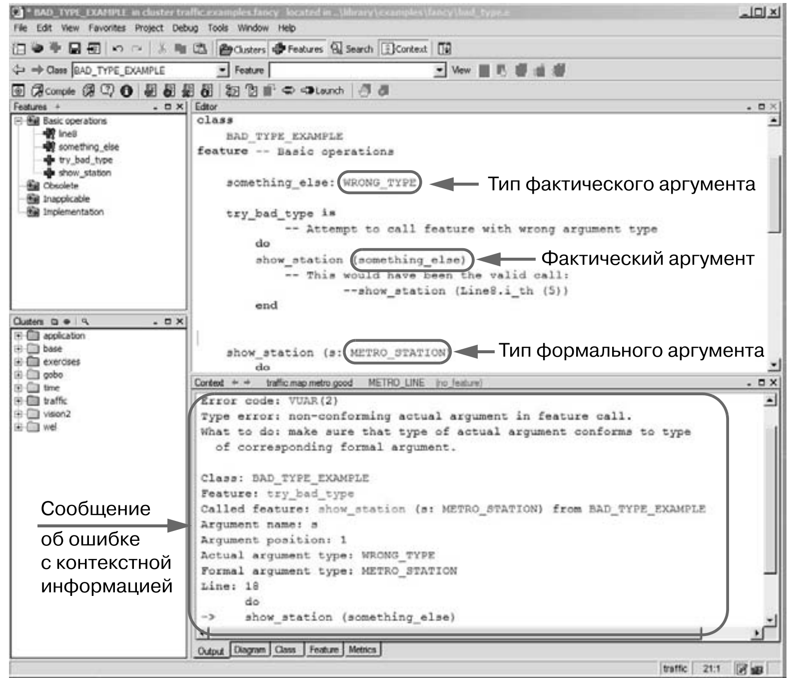


Рис. 8.1.

В этом примере мы передаем фактический аргумент произвольного типа *WRONG_TYPE* при вызове метода *show_station*, чей формальный аргумент имеет тип *METRO_STATION*. Сообщение об ошибке объясняет, что сделано неверно.

В теле метода *show_station* мы используем формальный аргумент *s* как значение, обозначающее станцию. Операция, применяемая к *s*, при любом вызове будет выполняться над фактическим аргументом: в нашем примере – над станцией, заданной *Line8.item*.

Не все методы имеют аргументы – примером является метод *remove_all_segments*.

Оставшаяся часть объявления *show_station* содержит следующие элементы.

- Метод должен иметь заголовочный комментарий, объясняющий, что он делает. В примере «— *Подсветить s в форме, согласованной со статусом*» стиль требует комментировать все формальные аргументы (здесь *s*), указывая роль каждого аргумента. Следует

избегать избыточности (скажите «*s*», но не «станция “*s”*», поскольку предыдущая строка объявляет *s*: *STATION*).
- Предусловие, здесь *s /= Void*, устанавливает, что работа возможна только с присоединенными фактическими аргументами.
- Предложение **do**, называемое телом метода, состоит из последовательности операторов, задающих алгоритм, реализуемый методом.
- Хотя в примере не появилось постусловие, но оно является возможной частью метода.

Интерфейс и реализация

В EiffelStudio можно видеть как реализацию, так и интерфейс метода, такого как *show_station*.
- Реализация (объявление) появляется в облике класса по умолчанию, известном как «текстовый облик». Это полное объявление метода.
- Интерфейс появляется по запросу «контрактного облика» при нажатии соответствующей кнопки. С его формой мы уже познакомились, когда изучали интерфейс методов класса *LINE*.

```
show_station (s: STATION)
        -- Подсветить s в форме, согласованной со статусом.
    require
        station_exists: s /= Void
```

Интерфейс метода адресован программистам, создающим клиентские классы. Из выше перечисленных элементов метода интерфейс содержит: сигнатуру, заголовочный комментарий, предусловие и постусловие, но не показывает тело метода, деталей реализации. Интерфейс метода должен говорить, *что* метод делает, но не *как* он это делает. Сигнатура и контракты, дополненные поясняющим комментарием, достаточны, чтобы выразить «что».

Контрактный облик класса отличается от текстового тем, что опускаются некоторые детали синтаксиса – ключевое слово **end**, необходимое в программах во избежание двусмысленности, но не требуемое в описании интерфейса.

8.5. Скрытие информации

Прием, когда программисту – клиенту метода (класса, любого другого модуля) – предоставляется описание интерфейса, включающего лишь подмножество свойств программного элемента, называется **скрытием информации**.

Скрытие информации – один из ключевых инструментов, позволяющих строить большие программные системы и успешно бороться с их сложностью: предоставлять клиентам каждого элемента *только то, что нужно* для его использования.

Несмотря на название, скрытие информации не означает запрет на знакомство с реализацией, так как Traffic и другие Eiffel библиотеки доступны в исходной форме. В EiffelStudio нетрудно познакомиться со всеми методами класса *LINE* и других Traffic-классов. Скрытие информации исходит из того, что *для использования* элемента *не требуется* знание его реализации.

Если для каждого используемого метода пришлось бы читать полный его текст, то количество информации превзошло бы все разумные пределы. Скрытие информации позволяет нам ограничиться лишь небольшой частью, и это наш лучший друг в постоянной борьбе со сложностью.

Не все библиотеки дают доступ к исходному коду, скрывая свои ноу-хау. Скрытие информации – это технический прием, не связанный с экономическими или политическими решениями, это лишь способ защиты от груды не относящихся к делу деталей.

Скрытие информации – это оружие не только в борьбе со сложностью, но и с нестабильностью. Одно из главных свойств ПО – способность гибких изменений, – недаром его называют софтом. Всякое изменение программного элемента сказывается на его клиентах, у которых также есть клиенты, что может включать целую цепь изменений. Но для хорошо спроектированных элементов с тщательным отбором того, что составляет интерфейс элемента, многие изменения могут быть связаны только с реализацией, не затрагивая интерфейс, а, следовательно, не влияя на клиентов. Это дает неоценимый инструмент для управления программными проектами.

Современная ОО-технология с наследованием и динамическим связыванием делает следующий шаг в скрытии информации, позволяя клиентам не только не вникать в детали применяемых к объектам операций, но и не знать точные типы этих объектов. Но до этого этапа предстоит еще дойти, а пока ограничимся скрытием реализации методов.

Время программирования!

Эксперименты в EiffelStudio и скрытие информации.

При нажатии кнопки «Compile» EiffelStudio не компилирует всю систему заново, что могло требовать большого времени: компилируются только те классы, которые были модифицированы после последней компиляции, и те, которые прямо или косвенно зависят от них. Это называется *нарастающей* (*incremental*) *компиляцией*. В студии она выполняется *автоматически*, нет необходимости в указании модифицированных классов, в указании зависимостей между классами.

Скрытие информации является основой анализа зависимостей: если вы изменили только реализацию, EiffelStudio обнаружит это и не будет перекомпилировать клиентские классы. Если же изменения затронули интерфейс, то придется перекомпилировать и клиентов. Вы можете следить за этим процессом.

1. Добавьте метод *r* в класс *LINE*. Не важно, что делает этот метод, пусть лишь у него будет аргумент и предусловие.
2. В методе *traverse* из класса *ROUTES* добавьте вызов *r*. Убедитесь, что вызов корректен, – он должен при вызове задавать фактический аргумент требуемого типа.
3. Перекомпилируйте систему. Заметьте, какие классы были скомпилированы заново (для просмотра результатов компиляции выберите вкладку «Output»).
4. Внесите изменения в тело *r*, не изменяя его интерфейс. Перекомпилируйте, наблюдая за процессом компиляции.
5. Теперь добавьте предложение постусловия в *r*, что изменяет его интерфейс. Перекомпилируйте, проследив за процессом компиляции *ROUTES*.
6. Для возвращения системы в первоначальное состояние удалите *r* из *LINE* и вызов *r* из *traverse*. Перекомпилируйте и убедитесь, что все работает должным образом.

8.6. Процедуры против функций

Есть два вида методов:

- **Процедура**: выполняет некоторые действия. Вызов процедуры в вызывающем методе является оператором. Методы *traverse* и *show_station* являются примерами процедур. К процедурам относятся и изучаемые ранее *процедуры создания*, служащие для создания и инициализации объектов класса.
- **Функция**: вычисляет некоторое значение. Вызов функции в вызывающем методе представляет *выражение*. С реализацией функций нам пока не пришлось встретиться, но вызывать функции приходилось: запрос *i_th* в классе *LINE* является функцией.

Разница известна:

- процедура реализует *команду*;
- функция реализует *запрос*.

Команды могут быть реализованы только процедурами, но запросы могут быть реализованы как функциями, так и атрибутами, о чем пойдет разговор в следующей главе.

Мы видели, что **сигнатура** процедуры характеризуется списком типов формальных аргументов, как при объявлении метода *show_station*:

```
show_station (s: STATION)
```

Сигнатура функции дополнительно должна указывать тип значения, возвращаемого функцией. Это можно видеть на примере запроса *i_th* в *LINE*, возвращающего результат типа *STATION*:

```
i_th (i: INTEGER):STATION
```

Оставшаяся часть объявления функции имеет те же элементы, что и процедура: заголовочный комментарий, предусловие, постусловие, тело. В теле функции и в постусловии необходимо имя для именования результата, возвращаемого функцией, – для него зарезервировано специальное ключевое слово **Result.**

8.7. Функциональная абстракция

Методы являются базисными алгоритмическими блоками, входящими в наши классы, а через них и в системы.

Используйте методы как механизм **алгоритмической абстракции**. Абстракция означает концентрацию на сущности, а не на деталях, на общих концепциях, а не на отдельных примерах. Абстракция почти всегда влечет **именование**. Как только выделена полезная абстракция, дайте ей имя, что позволит ссылаться на нее в будущем. В программировании мы встречаемся с двумя полезными видами абстракций.

- **Абстракцией данных**, которую дает нам *класс,* представляющий описание данных программы – объектов.
- **Абстракцию алгоритмов,** называемую также **функциональной абстракцией**, которая позволяет описать абстракцию, стоящую за нашими алгоритмами.

Для сохранения управляемости системой, даже в тех случаях, когда алгоритмы включают много деталей, используются методы. Оба подхода при проектировании метода «сверху вниз» и «снизу вверх» представляются привлекательными.

- Создав алгоритмический элемент, покрывающий важный шаг процесса разработки, превратите его в метод. Тогда у него появится имя и точная спецификация (сигнатура, заголовочный комментарий и контракт). Это превратит его, наряду с другими преимуществами, в хорошо определенный программный элемент, допускающий повторное использование. Это соответствует подходу «снизу вверх».
- При разработке «сверху вниз» в процессе разработки идентифицируется метод, детализация которого временно не выполняется, что позволяет сосредоточиться на достижении главной цели.

В этой второй роли альтернативой методу может быть псевдокод. Мы встречались с использованием псевдокода в наших примерах:

```
– "Создать линию и заполнить ее станциями"
```

Псевдокод можно заменить методом, называемым «**заглушкой**» или **держателем места**. В этом случае систему можно скомпилировать, поскольку достаточно существования метода, даже если он ничего не делает. Вот пример:

```
create_fancy_line
        – Создать фиктивную линию fancy_line и заполнить ее станциями
    do
        – Здесь следует указать (ваше имя, текущую дату)
    ensure
        line_exists: fancy_line /= Void
    end
```

Обратите внимание, постусловие задает часть контракта: метод должен создать объект и связать его с *fancy_line*.

Почувствуй методологию

Держатели места – заглушки методов

Если вы используете заглушку, всегда включайте в нее информацию о себе и о дате создания, а также заголовочный комментарий и другие пояснения того, что вы собираетесь позднее делать. Если для метода можно задать контракт (предусловие и постусловие), то напишите его сразу, включив в заглушку. Контракт, являясь частью спецификации метода, позволяет понять, что нужно делать, и будет служить руководством, когда вы вернетесь к детализации метода, превращая заглушку в настоящий метод.

8.8. Использование методов

Методы – алгоритмические абстракции – лучшее средство в борьбе со сложностью. Используйте их при разработках «снизу вверх», подготавливая существующие элементы для дальнейшего повторного использования. Используйте их при разработках «сверху вниз», подготавливая элементы, которые еще предстоит реализовать.

Программисты всегда заботятся об *эффективности*, в частности, о скорости выполнения. По этой причине они могут избегать создания метода, зная, что всякий вызов метода требует дополнительных расходов времени. *Хорошие* программисты также заботятся об эффективности, но они заботятся и о *качестве* ПО. Есть три причины, по которым крайне редко следует ограничивать себя в создании методов.

- Архитектура современных компьютеров позволяет резко снизить накладные расходы на вызов метода.
- За исключением случаев, когда вызов появляется во внутреннем цикле и выполняется многократно, доля расходов на вызов пренебрежимо мала в сравнении с общим временем работы, по крайней мере, для программ, выполняющих интенсивные вычисления (если общее время работы мало, то нет и особого смысла говорить об эффективности). Обычно имеет смысл заботиться о критических участках программы, на долю которых приходятся основные затраты времени.
- Не стоит самому беспокоиться даже о критически важных методах, для которых потери на вызов могут быть ощутимы. Можно положиться на компилятор EiffelStudio, который способен в режиме «оптимизации» выполнять автоматическое встраивание метода в точки вызова. Студия позволяет управлять этим процессом, задавая, например, максимальный размер метода, для которого допускается преобразование его во встраиваемый (in line) метод.

Изучение сложности алгоритмов даст нам лучшие рамки для обсуждения эффективности, выражая производительность как функцию от размера множества данных.

8.9. Приложение: Доказательство неразрешимости проблемы остановки

Ранее было отмечено, что невозможно создать алгоритм («эффективную процедуру»), который бы для любой программы определял, остановится ли она. Давайте докажем этот факт, предполагая, что, если бы такой алгоритм существовал, то мы могли бы написать его реализацию на Eiifel.

Предположим, что алгоритм существует и мы написали реализующую его функцию:

```
terminates (root_directory: STRING): BOOLEAN
        - Для программной системы, заданной аргументом root_directory,
        - если она есть, определить, завершится ли она?
    do
            … Подходящий, очень умный алгоритм …
    end
```

Аргумент *root_directory* задает имя каталога, в котором хранится «ECF» программы – описание системных установок, дающих доступ ко всем классам, спецификациям корневого класса и корневой процедуры создания. Для простоты будем полагать, что ECF – это файл, названный *system.ecf*, хранящийся в каталоге. Наша способность решить проблему остановки означает, что мы способны написать «Подходящий, очень умный алгоритм», так что *terminates* (*r*) будет возвращать **True**, если и только если такой файл задан аргументом *r* и выполнение соответствующей программы приводит к завершению.

Можно изменить соглашения, адаптируя доказательство к любому другому языку программирования. Вместо ECF аргумент мог быть просто именем файла, содержащего текст всех классов программной системы, с указанием корневой процедуры и класса. Что важно на самом деле – это то, что аргумент функции *terminates* должен позволять передать ей текст некоторой системы. Работа функции *terminates* состоит в определении, будет ли переданная ей система останавливаться.

До сих пор мы полагали, что проверяемой системе не нужен никакой вход во время ее работы. В более общей постановке можно задать и входные данные:

```
terminates_on_input (root_directory: STRING; input: STRING): BOOLEAN
       – Для программной системы, заданной аргументом root_directory,
       – если она есть, определить, завершится ли она на входе input?
    do
       … Подходящий, очень умный алгоритм …
    end
```

Будем рассматривать первую форму постановки задачи, но доказательство применимо и к форме со входными данными.

Доказательство простое. Если написан метод *terminates*, то тогда нетрудно написать и программу со следующей корневой процедурой:

```
paradox
           – Завершается, если и только если не завершается terminates.
       do
           from
           until
              not terminates ("C:\your_project")
           loop
           end
       end
```

Здесь C:\your_project – это произвольное имя каталога (папки) в стиле Windows. Фактически важно лишь то, что мы указываем здесь каталог, хранящий саму систему «paradox». Тогда вызов *terminates* решит, завершится ли эта система. Давайте рассмотрим, как будет выполняться процедура создания.

- Если функция *terminates* определит в результате анализа текста системы «paradox», что ее выполнение **не завершается**, то в этом случае условие выхода из цикла выполнится после первого выполнения тела цикла и система «paradox» тут же **завершит** свою работу. Пришли к противоречию с результатом, выданным «очень умным» алгоритмом *terminates*.
- Если функция *terminates* определит в результате анализа текста системы «paradox», что ее выполнение **завершается**, то в этом случае условие выхода из цикла никогда не будет выполняться и система «paradox» **зациклится**. Пришли опять-таки к противоречию с результатом, выданным «очень умным» алгоритмом *terminates*.

Это и доказывает, что невозможно написать функцию *terminates*, на вход которой можно было бы передать описание любого метода для определения его завершаемости.

Мы увидим в последующих главах более краткую версию доказательства, применяющую рекурсию. В одном из упражнений предстоит написать короткое доказательство, использующее агенты.

8.10. Дальнейшее чтение

Рис. 8.2. Дэвид Парнас (2007)

David Parnas: A Technique for Software Specification with Examples, in Communications of the ACM, vol. 15, no. 5, 1972, p. 330-336, and On the Criteria to be Used in Decomposing Systems into Modules, ibid, vol. 15, no. 12, 1972, p. 1053-1058.

На русском языке: «Метод спецификации модулей ПО с примерами» в сб. «Данные в языках программирования» под ред. В. Агафонова, М., Мир. 1982 г.

Две классические статьи, в которых вводится понятие скрытия информации. До сих пор не утратили значения.

8.11. Ключевые концепции, изученные в этой главе

- Методы обеспечивают функциональную абстракцию: способность именования возможно параметризованных алгоритмов.
- В подходе «сверху вниз» метод может выступать в роли «заглушки» – держателя места, для алгоритма, который будет детализирован позже в процессе проектирования. В подходе «снизу вверх» уже созданный метод может повторно использоваться в разных проектах.
- В ОО-контексте методы являются одним из двух видов характеристик класса (feature). Сами методы разделяются на две категории: функции, возвращающие результат, и процедуры, которые этого не делают.
- Методы могут иметь аргументы, которые позволяют вызывающему методу передать информацию, специфическую для каждого вызова.
- Метод имеет имя, сигнатуру, определяемую типами аргументов, результат, если он есть, контракт и тело, описывающее алгоритм.
- Имя, сигнатура и контракт определяют интерфейс метода, доступный авторам клиентских модулей.
- Скрытие информации является механизмом, позволяющим отделить интерфейс от реализации, что позволяет клиентам применять методы, основываясь только на информации, заключенной в интерфейсе.

- Скрытие информации облегчает разработку больших систем, повторное использование программных элементов и гладкую эволюцию системы.

Новый словарь

Actual argument	Фактический аргумент	**Body**	Тело
Data abstraction	Абстракция данных	**Declaration**	Объявление
Formal argument	Формальный аргумент	**Function**	Функция
Functional abstraction	Функциональная абстракция	**Implementation**	Реализация
Information hiding	Скрытие информации	**Incremental compilation**	Возрастающая компиляция
Placeholder routine	Заглушка – держатель места	**Procedure**	Процедура
Routine	Подпрограмма, метод	**Signature**	Сигнатура

8-У. Упражнения

8-У.1. Словарь09

Дайте точные определения терминам словаря.

8-У.2. Концептуальная карта

Добавьте новые термины в концептуальную карту, созданную в предыдущих главах.

9

Переменные, присваивание и ссылки

Программы используют имена или сущности для обозначения значений периода выполнения. Отличительным свойством большинства программ является то, что некоторые сущности, называемые «переменными сущностями» или просто переменными, могут обозначать значения, изменяющиеся во время выполнения. Предыдущие примеры неявно исходили из этого предположения, хотя базисная операция – присваивание – до сих пор формально еще не введена.

Эта концепция, обманчиво простая с первого взгляда, полна удивительных следствий. Мы будем изучать ее в этой главе наряду с несколькими связанными приемами, в частности, с использованием *ссылок*, определяющих структуру объектов в период выполнения.

Математика – статична, ПО – динамично

Способность программ изменять собственное окружение представляет наиболее важное отличие конструирования ПО от математического вывода – двух видов деятельности, во многом схожих в других отношениях.

Математики используют преобразования, но они являются механизмом описания одних значений в терминах других, не изменяя при этом значений уже существующих объектов. Если я пишу «*Пусть $y = cos(x)$*», я ничего не изменяю и даже не создаю, просто даю имя значению косинуса от x, которое существует само по себе, не беспокоясь, собирается ли кто-либо говорить о нем.

Даже, если я потом говорю: «Предположим теперь, что *y принимает значение $sin(x)$*», – то для удобства имя y повторно используется, но обозначает совсем другой математический объект. Когда математик описывает последовательность «Пусть f_1 и f_2 равны 1, и $f_{i+2} = f_i + f_{i+1}$ для каждого $i > 0$», то речь идет о бесконечной последовательности чисел. У программиста, работающего с числами Фибоначчи, скорее всего, появятся две переменные, меняющие свои значения.

В программировании мы не описываем результаты заданием свойств, которым они должны удовлетворять. Мы должны **вычислить** результаты по некоторому алгоритму, используя компьютер и его память.

При выполнении алгоритма последовательно вычисляемые значения записываются в память. Если бы память была бесконечно большой и бесконечно дешевой, то можно было бы для каждого нового значения выделять новую ячейку. Но память – хотя и большая, но конечная, поэтому приходится повторно использовать одни и те же ячейки, в которых будут храниться новые значения, когда старые уже не нужны.

Поэтому в программировании *переменные*, в отличие от своих двойников в математике, соответствуют своему имени, поскольку изменяют свои значения во время выполнения. Присутствие таких изменений – один из главных вызовов в нашем стремлении вывести свойства программ, используя базисные инструменты, имеющиеся в нашем распоряжении: логику, вообще математику.

Степень различия математики и программирования различна. Функциональное программирование и поддерживающие его функциональные языки вводят конструкции, близкие к математическому выводу, исключая или строго ограничивая изменения и побочные эффекты в процессе вычислений. Базисной конструкцией является функция в ее математическом смысле без побочных эффектов. В одной из последующих глав эта проблема обсуждается подробнее и рассматриваются основные конструкции функциональных языков.
Eiffel принадлежит к классу императивных языков. Благодаря принципу разделения команд и запросов побочный эффект разрешается только в процедурах, что облегчает выводы о программах в стиле, подобном математике.

9.1. Присваивание

Присваивание – это оператор, разрешающий изменять значение переменной.

Для примеров и упражнений этой главы следует использовать новый класс, названный *ASSIGNMENTS*.

Суммируем время поездки

Следующая простая задача будет служить примером: зная среднее время поездки между двумя соседними станциями, вычислить общее среднее время, требуемое на поездку в метро от начальной до конечной станции линии. Добавим в класс *LINE* функцию *total_time*, занимающуюся этим вычислением.

Алгоритм прямолинеен: последовательно проходя остановки на линии, следует к общему времени добавлять время до предыдущей остановки. Необходимая информация может быть получена по запросу из класса *STOP*:

```
time_to_next: REAL
        -- Ожидаемое время проезда до следующей остановки: от отправления до
        -- отправления,
        -- исключение для последней остановки: от отправления до прибытия.
    require
        has_next: is_linked
```

Предусловие метода *is_linked* требует, чтобы остановка была связана со следующей (не была последней).

Наша желаемая функция *total_time* имеет следующую общую форму:

```
total_time: REAL
            -- Оценка времени поездки по всей линии
        do
```

```
        from
            start
            – “Установить Result в ноль”
        invariant
            – “Значение Result – это время поездки от первой станции
            – до текущей станции, заданной позицией курсора”
        until
            is_last
        loop
            – “Увеличить Result на время до следующей станции”
            forth
        variant
            count - index
        end
    end
```

Переменная **Result** обозначает результат, возвращаемый функцией. Две команды псевдокода будут заменены присваиваниями.

Булевская функция *is_last* говорит нам, установлен ли курсор на последней станции. Заметьте разницу между схемой цикла, рассмотренной в предыдущей главе, где в конце цикла курсор имел значение *after*, а не *is_last*.

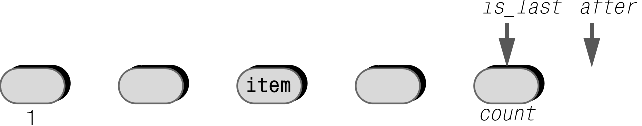


Рис. 9.1. Запросы к базисной форме списка

Время теста

Когда выходить из цикла

Почему цикл для *total_time* использует *is_last* в качестве условия выхода, а не обычный *after*?

(*Подсказка*: сравните число остановок с числом интервалов. Сравните также выражение «variant» для двух циклов)

Команды псевдокода должны обновлять значение **Result**. Присваивание предназначено именно для этого.

Оператор присваивания имеет вид:

```
target := source
```

Здесь *source* – это выражение, а *target* – переменная, такая как **Result**. Левая часть присваивания *target* называется *целью* присваивания, а правая часть – *source* – *источником*. Эффект выполнения состоит в замене значения *target* на *source*. Чтобы быть точным:

Почувствуй семантику

Эффект присваивания

Выполнение оператора присваивания *target:= source* состоит из:

A1. вычисления (computing) значения выражения *source*;

A2. переменная *target* получает это значение, начиная с момента присваивания, и сохраняет его до очередного присваивания *target*.

Это единственный эффект оператора. В частности, это никак не отражается на *source*.

Если вы программировали до чтения этой книги и хорошо знакомы с присваиванием, то, возможно, сочтете это определение педантичным. Но следует быть точным. Новички в программировании, встречаясь с присваиванием *x:= y*, иногда интуитивно воспринимают присваивание как передачу денег: если *y* передал свое значение *x*, то сам *y* теряет его и получает значение по умолчанию. Ничего подобного, *y* остается неизменным.

Выражение, такое как *source*, обычно включает переменные; «вычисление» его (A1) будет использовать их значения, полученные как результат предыдущих присваиваний.

Используем присваивание для уточнения функции нашего примера:

```
total_time: REAL
        -- Оценка времени проезда по всей линии.
    do
        from
            start
            Result := 0.0
        invariant
            -- "Значение Result – это время поездки от первой станции
            -- до текущей станции, заданной позицией курсора"
        until
            is_last
        loop
            Result := Result + item.time_to_next
            forth
        variant
            count - index
        end
    end
```

На каждом шаге цикла мы добавляем к текущему значению **Result** время до следующей станции. Так как мы также выполняем *forth*, инвариант цикла сохраняется. При выходе из цикла инвариант скажет, что **Result** задает время от первой станции до станции в позиции курсора, но так как теперь *is_last* истинно, **Result** дает общее время проезда по линии.

Время программирования!

Оцените время проезда по линии метро

Напишите функцию total_time8 для вычисления и показа времени проезда по 8-й линии метро. Используйте вышеприведенную модель, но не изменяйте класс *LINE*. Новую функцию сделайте частью класса *ASSIGNMENTS*, применив ее к *Line8*.

Локальные переменные

В предыдущей главе мы видели схему алгоритма для вычисления значения максимума множества значений. В отсутствие присваивания использовались элементы псевдокода:

```
– “Определить max равным N1”
– “Определить i равным 1”
– “Переопределить max как большее из текущего максимума и Ni+1”
– “Увеличить i на единицу”
```

Мы можем теперь выразить этот алгоритм, используя присваивание. Но давайте напишем функцию, вычисляющую «максимальное» (в лексикографическом порядке) имя среди имен всех станций линии:

```
highest_name (line: LINE): STRING
        – Последнее (в алфавитном порядке) имя станции на линии
    require
        line_exists: line /= Void
    local
      i: INTEGER
        new: STRING
    do
        from
            Result := line.south_end.name
            i := 1
        invariant … -- Как ранее
        until
            i = line.count
        loop
            new := i_th (i).name
            if new > Result then
                Result := new
            end
        i := i + 1
end
```

Присваиваний в этом примере немало. В предложении **from** переменная **Result** инициализируется именем первой станции *south_end*, а переменная *i* получает значение 1. Затем в цикле, если имя текущей станции, обозначенное как *new*, больше, чем текущий максимум, то значение **Result** меняется на *new*. Корректность этого алгоритма зависит от двух свойств, выраженных инвариантами соответствующих классов:

- у *LINE* всегда есть хотя бы одна станция, доступная как *south_end* или, эквивалентно, *i_th* (1);
- каждая станция метро имеет имя *name*, отличное от void.

Заметьте, сравнение строк использует алфавитный порядок, его еще называют лексикографическим порядком: *s2* > *s1* имеет значение *True*, если и только если *s2* следует за *s1* в алфавитном порядке.

Время программирования!

Наибольшее в алфавитном порядке имя станции

Добавьте функцию *highest_name* в пример класса для этой главы – *ASSIGNMENTS*, и используйте ее для показа «максимального» в алфавитном порядке имени станции 8-й линии метро.

Принципиальной новинкой этого примера является использование *локальных переменных*. Рассмотрим объявление:

```
local
        i: INTEGER
        new: STRING
```

Здесь вводятся две сущности, *i* and *new*, которые метод может использовать для хранения промежуточных результатов, требуемых алгоритму. Локальные переменные и являются такими сущностями – локальные для метода и вводимые ключевым словом **local**. Можно было бы обойтись без локальных переменных, вводя их как *атрибуты* класса, о которых будем подробно говорить в этой главе. Но локальные переменные не заслуживают такого статуса. Атрибуты класса – это характеристики класса, его компоненты (feature), которыми обладают все экземпляры класса. Здесь же сущности *i* и *new* необходимы временно для каждого исполнения метода. Когда выполнение метода завершается, *i* и *new* выполнили свое дело и могут уйти.

Имена локальных переменных могут быть произвольными, лишь бы они не стали причиной конфликта имен.

Правило локальных переменных

Локальная переменная не может иметь имя атрибута или метода класса, в котором она находится. Она не может иметь имя аргумента метода, в котором она находится.

В принципе, допустимо совпадение имен локальных переменных с компонентами класса при условии, что внутри метода конфликтующее имя означает локальную переменную, но такое соглашение может быть источником недоразумений и ошибок. Имена недорого стоят – когда вам нужна новая переменная, выбирайте для нее новое имя.

Заметьте, ничто не мешает использовать одни и те же имена локальных переменных для разных методов в одном и том же классе (некоторые имена – *item*, *count*, *put* ... – встречаются многократно во многих различных классах). Такие случаи не приводят к двусмысленностям, так как омонимы появляются в разных контекстах – в разных областях видимости переменных.

Результат функции

Как показано в двух последних примерах, **Result** может появляться в функциях для обозначения результата, вычисляемого функцией. Напоминаю, процедуры и функции являются двумя видами методов класса. Функции вычисляют и возвращают значение, **Result** служит для обозначения этого значения в тексте функции. Процедуры, в отличие от функций, могут изменять объекты, но не возвращают результат. Очевидно, что **Result** не может использоваться в процедурах.

Рассмотрим вызов функции в методе класса *ASSIGNMENTS*:

```
Console.show (highest_name (Line8))
```

Здесь вызывается функция *highest_name* и отображается результат ее вычисления, который является последним значением **Result**, полученным в ходе вычисления функции непосредственно перед ее завершением. Вы могли наблюдать за этим в процессе выполнения последней сессии «Время программирования».

Формально **Result** является локальной переменной. Единственное отличие: он не объявляется в теле функции, а автоматически доступен для любой функции и неявно объявлен как переменная, тип которой совпадает с типом возвращаемого функцией значения, например, *REAL* для *total_time*, *STRING* для *highest_name*.

Это также обозначает, что **Result** является **резервируемым словом**, которое нельзя использовать для собственных идентификаторов программы.

Определение: резервируемое слово

> **Резервируемое слово** – это идентификатор, играющий специальную роль в языке программирования, и как следствие, он не может использоваться для других целей в конкретной программе – обозначать имена классов, методов, переменных и прочего.

Резервируемые слова обобщают понятие ключевого слова, введенного ранее. Пример с **Result** иллюстрирует, почему ключевые слова являются только одним из двух видов резервируемых слов.

- Ключевые слова – **class**, **do**... – играют только синтаксическую роль; они не обозначают значение в период выполнения.
- Другие резервируемые слова, такие как **Result,** имеют значение и семантику. Другими примерами резервируемых слов являются **True** и **False**, обозначающие булевские значения.

Обмен значениями (своппинг)

Рассмотрим типичный пример с присваиванием и локальной переменной. Предположим, что переменные *var1* и *var2* имеют один и тот же тип *T*. Пусть необходимо, чтобы переменные обменялись значениями. Следующие три оператора выполняют обмен:

```
swap := var1; var1 := var2; var2 := swap
```

Обмен требует третью переменную *swap*, типично объявляемую как локальную переменную типа *T*. Схема свопинга такова:

- первое присваивание сохраняет начальное значение *var1* в *swap*;
- второе присваивание изменяет значение *var1* на начальное значение *var2*;
- третье присваивание изменяет значение *var2* на значение *swap*, хранящее начальное значение *var1*.

Понятно, почему нам необходима переменная *swap*: нам нужна память для сохранения значения одной из переменных, прежде чем оно будет изменено. Заметьте, что важен порядок выполнения операторов при обмене, хотя он не единственно возможный (переменные *var1* и *var2* можно поменять местами ввиду симметричности обмена).

Переменные, такие как *swap*, используемые для промежуточных целей, известны как **временные переменные**, типично они объявляются как локальные переменные метода.

Мощь присваивания

Символ присваивания – «:=». Для присваивания *i*:= *i* + 1 обычно говорят: «*i* получает значение *i* + 1» или «*i* присваивается значение *i* + 1».

Эффект состоит в замене значения *target* на значение выражения *source*. Прежнее значение *target* будет потеряно навсегда: никакие записи не сохраняются. Присваивание является двойником (на более высоком уровне языка программирования) машинной операции компьютера – записи значения в ячейку памяти. Так что, если значение некоторой переменной вам может понадобиться в будущем, то, прежде чем изменить его, присвойте это значение другой переменной!

Рассмотрим часто встречающийся образец, когда источник присваивания (выражение) содержит в качестве одной из переменных цель присваивания. Этот образец встречался в примерах обоих рассмотренных нами методов:

```
Result := Result+ item.time_to_next
i := i+ 1
```

Новое значение цели вычисляется на основе ранее вычисленного значения и новой информации. Эта схема близка к стандартной математической схеме определения последовательности значений рекуррентными соотношениями. Вот как выглядит слегка упрощенная версия примера, рассмотренного в начале главы:

```
"Пусть дано s0, тогда si+1= f (si) для каждого i ≥ 0"
```

Здесь *f* некоторая функция (в примере с числами Фибоначчи функция *f* зависела от двух ранее вычисленных значений, а не от одного). Для вычисления s_n для некоторого *n* >= 0 можно использовать цикл:

```
from
    Result := "Заданное начальное значение S0"
    i := 0
invariant
    "Result = si"
until
    i = n
variant
    n - i
loop
    i:=i+1
    Result:=f(Result)
end
```

Эта схема применима только в том случае, если нет необходимости в хранении всех элементов последовательности и достаточно знать только последнее значение s_i на каждом шаге. В обоих примерах так и происходит.

Убедитесь, что вы понимаете разницу между математическим свойством $s_{i+1} = f(s_i)$ и оператором присваивания $x := f(x)$, в котором заключен механизм *изменений*, свойственный ПО. Этот механизм чужд математике и усложняет выводимость свойств программ. Заметьте, в частности, разницу между оператором

```
x := y
```

и булевским выражением

```
x = y
```

используемом, например, в условном операторе **if** $x = y$ **then** .... Булевское выражение имеет те же свойства, что и равенство в математике, – оно **дескриптивно** (описательно), представляя возможное свойство(**true** или **false**) двух значений x и y. Оператор присваивания **императивен** (повелителен) – он предписывает изменить значение переменной в результате вычислений. Вдобавок он **деструктивен** (разрушителен), уничтожая предыдущее значение этой переменной.

Замечательным примером этой разницы является часто встречающийся в циклах оператор

```
i := i + 1
```

Булевское выражение $i = i + 1$ вполне легально, но бессмысленно, поскольку всегда имеет значение false.

Почувствуй синтаксис

Присваивание и равенство – неразбериха.

Первый широко применяемый язык программирования Fortran использовал для присваивания символ равенства «=». Это была оплошность. В последующих языках, таких как Algol и его последователи, для присваивания введена своя символика «:=», с сохранением символа равенства для операции эквивалентности.

По неизвестным причинам в языке C вернули знак равенства для присваивания, используя для эквивалентности «==». Это соглашение не только противоречит устоявшимся в математике свойствам ($a = b$ в математике означает то же, что и $b = a$), но и является постоянным источником ошибок. Если вместо **if** ($x == y$) … по ошибке написать **if** ($x = y$) …, то результат, допустимый в C, даст неожиданный эффект: x получит значение y, далее выработается булевское значение **False**, если значение y и новое значение x равно нулю, и **True** – в противном случае.

Такие современные языки, как C++, Java и C#, оставили C-соглашение для присваивания и равенства, хотя в двух последних случаях действуют более строгие правила, позволяющие избежать появления подобных ошибок при выполнении программы.

9.2. Атрибуты

Есть два вида переменных (сущностей, которым можно присваивать значения). Первый вид мы уже видели – это локальные переменные, включая **Result**. Второй вид, который начина-

ем изучать, – *атрибуты*. Он не вполне нов, неявно мы с ним встречались под маркой *полей* объекта, когда рассматривали создание объекта. Но теперь мы можем дополнить наше понимание этой концепции и указать место атрибутов среди сущностей и других созданий в нашем ОО-питомнике.

Поля, методы, запросы, функции, атрибуты

Мы уже видели при обсуждении создания объекта, что он существует во время выполнения в памяти компьютера как набор полей, часть из которых является ссылками, другие относятся к основным («развернутым») типам:

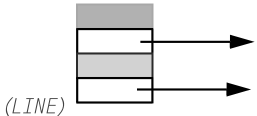


Рис. 9.2. Объект и его поля

Подобно любому другому свойству объекта, эти поля должны приходить из спецификации генерирующего класса. Действительно, каждое поле приходит из запроса, и даже более точно – атрибута.

Вернемся к началу. Метод, как вы знаете, является командой или запросом. Запрос, в отличие от команды, возвращает результат. Запрос может, в свою очередь, быть либо функцией, либо атрибутом. Это функция, если результат получается в результате вычисления функции. Например, класс *LINE* имеет запрос:

```
south_end: STATION
        -- Конечная станция на южной стороне
    do
        if not is_empty then
            Result := metro_stops.first.station
        end
    end
```

Это функция. Но в том же классе существует другой запрос *без* алгоритма (**do** … **end** части):

```
index: INTEGER
        -- Индекс текущей станции на линии
```

Это атрибут. Включение его в класс означает, что каждый экземпляр класса будет иметь поле заданного типа – *INTEGER*, содержащее текущее значение *index* для станции:

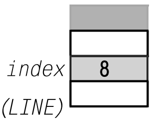


Рис. 9.3. Объект и его поля

Присваивание атрибуту

Как указано в комментарии, *index* в классе *LINE* – это индекс позиции «курсора». Курсор – это абстрактный механизм, позволяющий клиентам последовательно анализировать станции на линии, передвигаясь вперед и назад. Одной из команд, манипулирующей курсором, является команда *start*, устанавливающая курсор на первой станции (известной как *south_end*):

```
start
        – Установить курсор станции на первом элементе.
    do
        index := 1
        … Другие операторы…
    ensure
        on_first: index = 1
    end
```

Клиент может вызвать этот метод для конкретной линии, например:

```
Line8.start
```

Результатом этого вызова будет установка значения *index* для соответствующего экземпляра класса *LINE*. Если до вызова поле объекта имело значение 8, как на предыдущем рисунке, то вызов переустановит его в 1, не затрагивая другие поля объекта.

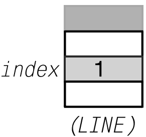


Рис. 9.4. Объект "Line"после вызова start

Вызов *Line8.start* является квалифицированным вызовом *start*, выполненный клиентом. Обычно возможен и неквалифицированный вызов *start*, выполняемый другими методами класса *LINE*.

Скрытие информации: модификация полей

Две другие процедуры класса также изменяют значение атрибута *index:*

```
forth
        –Передвинуть курсор к следующему элементу
    require
        not_after: not after
    do
        index := index + 1
    ensure
        moved_right: index = old index + 1
```

```
    end
go_ith (i: INTEGER)
        -- Передвинуть курсор к элементу с номером i
    require
        not_over_left: i >= 0
        not_over_right: i <= count + 1
    do
        index := i
    ensure
        set: index = i
    end
```

Все три процедуры позволяют клиентам устанавливать поле *index* для любого конкретного объекта, как например

```
Line8.start
Line8.forth
Line8.go_ith (5)                                                    [3]
```

Крайне важно понимать, что для клиентов такие вызовы процедур являются единственным способом модифицировать это поле. Не допускается для этих целей использовать присваивание – проверьте, если хотите, и посмотрите, что скажет компилятор в ответ на попытку:

```
Line8.index := 98                                                   [4]
```

Прежде всего, нарушен синтаксис присваивания: цель присваивания должна быть переменной, идентификатором, в то время как *Line8.index* является выражением.

Причина такой синтаксической защиты понятна. Позволить клиентам непосредственно модифицировать поля объектов поставщика означало бы игнорирование принципов скрытия информации и хорошего проектирования. В самом начале мы говорили, что объект следует воспринимать как машину, иллюстрируя это рисунком, повторно воспроизводимом ниже. Клиенты могут управлять этой машиной только через операции официального интерфейса – кнопки команд и запросов.

Выполнение непосредственного присваивания *your_machine, your_field:= my_value* эквивалентно вскрытию машины и ковырянию в ее внутренностях. Для реальных машин это означало бы лишение гарантий, для программной машины – лишение интерфейса и связанных с ним гарантий контракта.

Рис. 9.5. Объект “Line” как машина

Заметьте ключевую разницу между ошибочным присваиванием **[4]** и вызовом процедуры **[3]**. Вызов связан предусловием *go_ith*, устанавливающим:

```
require
    not_over_left: i >= 0
    not_over_right: i <= count + 1
```

Присваивание, в случае его разрешения, игнорировало бы предусловие.

Любая операция, которая может иметь доступ к полям объектам, их модификации, должна пройти через интерфейс, обеспечиваемый классом. Когда вы проектируете класс, ваша право и ваша обязанность – решить, что позволяется делать клиентам при работе с объектами класса. Для любого из атрибутов некоторого класса *T* вы можете разрешить клиентам устанавливать значения поля, но в этом случае вы должны определить в классе соответствующую процедуру, называемую сеттером (*setter*), или установщиком[1].

```
set_a (x: T)
        – Установить значение a равным x.
    do
        a := x
    ensure
        set: a = x
    end
```

Эту процедуру клиенты могут использовать без всяких ограничений *their_object.set_a* (*their_value*). Но можно ввести предусловие, как в *go_ith*, которое ограничивает допустимые значения. Можно ограничить клиента и другими способами: если из класса *LINE* удалить метод *go_ith*, то клиенты могли бы изменять поле *index,* только используя методы *start* и *forth*. Наконец, можно вообще не давать клиентам никакого метода, позволяющего им присваивать значения полю *index*.

В первом случае, когда при проектировании класса решено предоставить клиентам возможность модифицировать значение поля, некоторые полагают, что синтаксис присваивания **[4]** предпочтительнее вызова сеттера **[3].** Можно ли использовать синтаксис **[4],** но не как присваивание, а как синтаксический сахар, упрощенную форму вызова **[3]**?

Это и в самом деле возможно, если объявить процедуру сеттера как **команду присваивания** для связанного запроса, *a* или *index*. Для этого достаточно изменить объявление запроса следующим образом: *a*: *T* **assign** *set_a* или *index*: *INTEGER* **assign** *go_ith*. Тогда запись *obj.a:= v* является синтаксически корректной, но означает не присваивание, а вызов *obj.set_a* (*v*). Другие детали будут даны в дальнейших обсуждениях.

Независимо от синтаксиса, статическое семантическое правило одно и то же: единственный способ модифицирования объекта извне – через процедуру «сеттер». Для осознания фундаментальности такого подхода рассмотрите следующие два события в эволюции системы.

---

[1] В таких языках, как C# и другие, такие методы со специальным синтаксисом называют методами – свойствами.

- В какой-то момент вы решили – хотя это и не было включено в первоначальную концепцию ПО, – что каждая модификация атрибута должна фиксироваться записью в базу данных (например «1-го мая в 7.55 температура аппарата была изменена на 22 °C»).
- Добавлены ограничения на возможные значения атрибута (например, температура аппарата должна находиться в пределах от -5° C до +30° C). Это ограничение может быть введено в виде предложения инварианта класса.

Для выполнения первой модификации достаточно добавить операторы (обновления базы данных) в процедуру сеттер. Вторая модификация – более деликатная, поскольку влечет добавление предусловия к сеттеру, следовательно, отражается на каждом клиенте, работающем с полем. Так как каждый клиент использует сеттер, выявить таких клиентов нетрудно, что позволяет обновить их в соответствии с новыми требованиями, принуждая выполнять предусловие.

Если бы допускалось прямое присваивание, то для вас наступили бы тяжелые времена, пришлось бы разбираться со всеми клиентами, разыскивая модификации полей. Хуже того, для новых клиентов было бы невозможно реализовать новую политику регистрации изменений в базе данных или проверять, что значения соответствуют установленным границам.

Обобщим это обсуждение введением простого принципа.

Почувствуй методологию

Принцип модификации атрибута

Единственным способом, которым клиент устанавливает значения полей объекта, должен быть вызов (в любой синтаксической форме) экспортируемых сеттер-процедур.

Это ключевое правило современного проектирования является непосредственным следствием принципа скрытия информации. При программировании на Eiffel это принуждение происходит естественно. Для других языков программирования, где применяются другие политики управления доступом к атрибутам, это должно стать важным методологическим правилом.

Скрытие информации: доступ к полям

Предшествующее обсуждение показывает, как можно модифицировать поля объекта, соответствующие атрибутам класса. Правила не запрещают клиентам получать доступ к полям, таким как *index*. Клиент может, например, использовать выражение *Line8.index*, чтобы узнать текущее значение поля:

```
Line8.start
Line8.forth
Console.show (Line8.index)
```

В результате значение 2 будет показано в окне консоли.

В других ситуациях может требоваться полное скрытие информации, с полным запрещением и модификации, и доступа к атрибуту. Например, *LINE* имеет метод *id_generator*, используемый для своих внутренних потребностей, который ни в какой форме не должен быть доступен клиентам. Для этого достаточно раздел **feature**, содержащий объявление атрибута, записать в следующей форме: **feature** {*NONE*}.

В этом случае все методы и атрибуты этого раздела станут недоступны клиентам. Если просматривать класс *LINE*, то можно увидеть предложение **feature**, стоящее непосредственно перед **invariant** предложением:

```
feature {NONE} – Инициализация
    id_generator: ID_GENERATOR
              – Внутренняя идентификация текущей линии
```

В результате выражение *Line8.id_generator* приводит к ошибке, если оно встречается у любого клиента (проверьте это и посмотрите на сообщение компилятора). Соответственно, этот метод не входит в документацию класса. Взгляните на контрактный облик класса *LINE* – вы не увидите никакого упоминания об *id_generator*. Этот метод можно встретить только в неквалифицированных вызовах в методах самого класса *LINE*. Например, процедура *extend* использует (проверьте это самостоятельно) присваивание:

```
i := id_generator.generated_id
```

NONE является именем специального класса. Когда мы будем изучать наследование, увидим его место в общем миропорядке.

9.3. Виды компонентов класса (features)

Теперь мы знакомы со всеми компонентами класса и способны понять их полную классификацию.

Точка зрения клиента

С позиций клиента доступны заданные в интерфейсе методы класса, которые могут быть запросами или командами. Метод может:

- возвращать результат, тогда он является **запросом**;
- не возвращать результат, но может изменить состояние целевого объекта, тогда он является **командой**.

В первом случае есть две возможности, зависящие от того, как автор класса реализовал запрос.

- Он мог выбрать память, тогда значение запроса будет храниться в соответствующем поле каждого объекта класса. Это означает реализацию запроса **атрибутом** класса. Как следствие, команды класса ответственны за изменение этого поля всякий раз, когда это необходимо, например, *forth* изменяет *index*.
- Вместо хранения запроса можно выбрать его *вычисление*. Всякий раз, когда понадобится значение, оно будет вычисляться в соответствии с заданным алгоритмом. В этом случае запрос реализуется **функцией**. Примером является функция *south_end*.

Эта классификация показана на следующем рисунке.

«Память» означает хранение значения вместо его вычисления. Слово «процедура» на этом уровне избыточно, являясь синонимом слова «команда».

Понятие «**запрос**» важно как общая категория для атрибутов и функций. Клиенту безразлично, как реализован запрос. Хотя различие отражается в тексте класса, но оно не проявляется в интерфейсе класса. Взгляните еще раз на контрактный облик класса *LINE* – вы

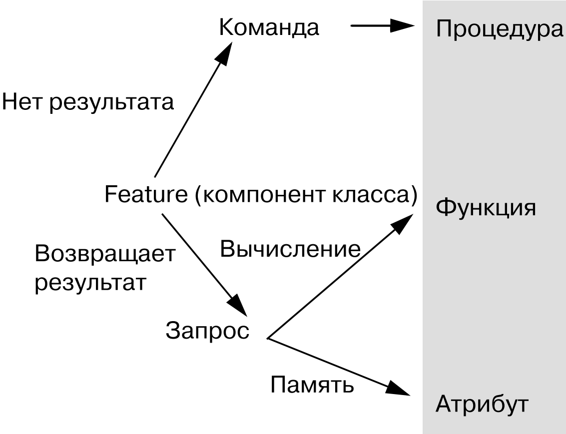


Рис. 9.6. Классификация: взгляд клиента

сможете увидеть следующие друг за другом (один в предложении —Access, другой – в —Measurement) интерфейсы двух запросов:

```
index: INTEGER
        – Индекс текущей станции на линии.
```

и

```
count: INTEGER
        – Число станций на этой линии.
```

Внешне они схожи. Но если посмотреть фактический текст класса, а не его контрактный облик, то *index* появляется как атрибут, в то время как *count* – это функция:

```
count: INTEGER
        – Число станций на этой линии.
    do
        Result := metro_stops.count
    end
```

Ничто в контрактном облике не указывает на эту разницу: для клиента – это запросы.

Политика, рассматривающая атрибуты и функции идентичными в контрактном облике класса, отражается в принципе разработки ПО.

Почувствуй методологию

Принцип унифицированного доступа

Когда клиенты класса используют запрос, нет никакой логической разницы, как он реализован – хранением в памяти или вычислением.

Атрибут задает хранение в памяти, функция – вычисление. То, что нет «логической разницы», не означает отсутствия «функциональной разницы». Разница может ощущаться в эффективности. Атрибуты требуют больше памяти, функции больше времени на их выполнение.

Выбор между двумя решениями всегда представляет извечный компромисс «память – время». Из этого вытекает важность принципа унифицированного доступа, разработчику вначале трудно выбрать лучшее решение. В ходе разработки и сопровождения проекта решение может меняться, иногда несколько раз. Принцип защищает клиента от этих изменений: нотация *some_object.some_query* применима в обоих случаях. Если бы доступ к атрибутам и функциям использовал различный синтаксис, то при изменениях пришлось бы обновлять существенную часть всей системы, что затрагивало бы всех клиентов данного класса.

Обсудим, как этот принцип связан с политикой скрытия информации.

- При проектировании класса разумно предоставлять клиентам доступ к атрибуту (чтение значения), особенно с учетом того, что они не знают, как реализован их запрос – атрибутом или функцией.
- Было бы **неправильно** позволять клиентам непосредственно присваивать значение атрибуту. Заметим, что среди прочего это позволило бы клиентам понять, что они имеют дело с атрибутом.

Позиция поставщика

Рассмотрим проблему классификации компонентов класса с позиций его разработчика:

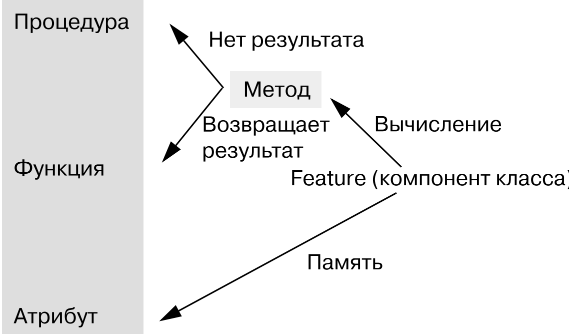


Рис. 9.7. Классификация: взгляд поставщика

Совмещая два рисунка, получим полную картину (рис 9.8).

Следует знать точное определение всех терминов, перечисленных на рисунке, их роль в построении классов, их использование клиентами класса.

Сеттеры и геттеры

Такие процедуры, как *set_a* или *go_ith*, устанавливающие значения атрибута, называются *сеттер-процедурами* или сеттер-командами.

В некоторых языках программирования также полезно создавать *геттер-функции*, чья единственная роль состоит в возвращении значения атрибута:

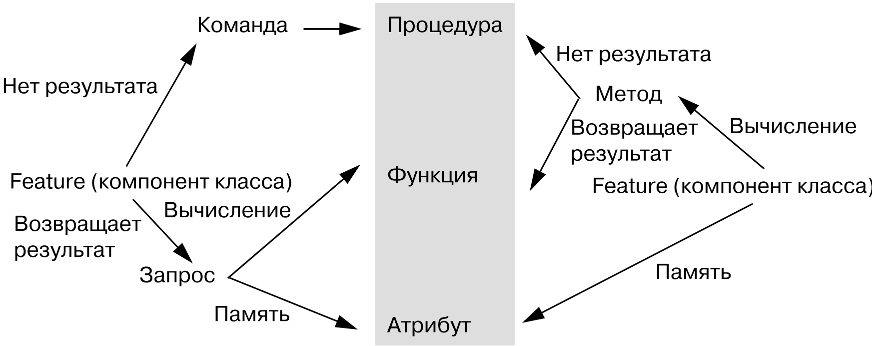


Рис. 9.8. Классификация: полная картина

```
current_index: INTEGER
        — Позиция курсора.
    do
        Result := index
    ensure
        same_as_attribute: Result = index
    end
```

Зачем это делать, если можно непосредственно использовать атрибут? Действительно, в рассматриваемых нами рамках в этом нет необходимости, Как мы видели, экспорт атрибута, такого как *index*, делает его доступным для клиентов со статусом «только для чтения». Это позволяет клиентам использовать значение *index* (как в *Line8.index*), не изменяя его, поскольку для изменений требуется сеттер-процедура. Экспортирование функции *current_index* приводит к тому же самому эффекту, что и экспортирование атрибута *index*.

Геттер-функции имеют смысл в таких языках, как C++, Java и C#, где экспорт атрибута означает его доступность и для чтения, и для записи, так что становится правильным присваивание *Line8.index:*= 98 **[4]**. Этот механизм чреват рисками, обсуждаемыми ранее, – он нарушает принцип скрытия информации и никогда не должен применяться. Поэтому стандартный совет методологии в разумных учебниках по этим языкам: не экспортировать атрибуты, закрывая их для клиентов. Вместо этого для предоставления клиентам доступа к атрибутам в зависимости от потребностей в классе создаются сеттер- и/или геттер-методы. В языках, таких как C#, есть понятие «метод – свойство», обеспечивающее такую возможность.

Разумные программисты, используя геттер-функции, достигают нужного эффекта скрытия информации в языках, не полностью отвечающих этому принципу. Если программист обнаруживает средство языка (в данном случае возможность чтения и записи для экспортируемых атрибутов) вместе со стандартным советом не применять это средство, то, мягко говоря, это бросает тень на проект языка. Программист может забыть совет и нарушить принцип скрытия информации. Применение совета означает написание геттер-функций, увеличивающих размер программного текста.

Предпочтительнее использовать языки, которые следуют свойству, обобщающему наше обсуждение.

Почувствуй методологию

> **Свойство экспорта атрибута**
> Экспорт атрибута, позволяющий клиентам получать доступ на чтение (но не запись) соответствующего поля объекта, вполне законен.
> Интерфейс класса не отличает (в отсутствие аргументов) экспортируемый атрибут от экспортируемой функции. Для клиентов оба вида появляются как *запросы*.
> Как следствие, отсутствует необходимость в геттер-функциях.

9.4. Сущности и переменные

Небольшая терминологическая чистка поможет нашему пониманию фундаментальных концепций, связанных с объектами и классами. Все, что касается методов и атрибутов из раздела **feature** мы уже пояснили. Но часто приходилось сталкиваться с такими терминами, как «*сущность*» и «*переменная*». Давайте и для них внесем ясность.

Базисные определения

Мы знаем, что сущность – это идентификатор, обозначающий возможное значение в период выполнения. Перечислим все возможные варианты этого понятия.

Определение: виды сущностей

> Сущностью может быть:
> E1: атрибут;
> E2: локальная переменная метода, включая предопределенную переменную **Result**;
> E3: формальный аргумент метода;
> E4: **current**, имя текущего объекта.

Если вы недоумевали, почему *index* иногда называли атрибутом, а иногда сущностью, то E1 объясняет причину – он и то, и другое. Чтобы быть совсем точным, идентификатор *index* является *сущностью*, для клиента он обозначает *запрос*, в реализации класса он задан *атрибутом*.

Как сущность, *index* является еще одной вещью: *переменной*. Сущности разделяются на два вида.

- Некоторые сущности могут изменять значение во время выполнения, выступая в роли цели присваивания, – это **переменные**. Они включают локальные переменные (E2) и один вид атрибутов (E1) – «переменные атрибуты».
- Другие будут сохранять значение во время выполнения, и они называются константными сущностями. Они включают формальные аргументы (E3), **Current** (E4) и второй вид атрибутов – «константные атрибуты».

Понятие переменной заслуживает собственного определения.

Определения: переменная, переменная сущность

> **Переменная сущность,** или просто **переменная,** является сущностью, чье ассоциированное значение может изменяться во время выполнения.
> Переменные включают *локальные переменные* и *атрибуты*.

Как обычно, локальные переменные включают **Result**.

Переменные и константные атрибуты

Атрибуты могут быть либо переменными, как во всех встречающихся до сих пор примерах, либо константами. Атрибуты, объявленные в обычной форме, являются переменными, например, *index* в объявлении:

```
index: INTEGER
```

Синтаксически **константный атрибут** распознается за счет того, что его объявление заканчивается знаком равенства, после которого следует значение. В классе *LINE* можно видеть (в разделе **feature** {*NONE*}) объявление:

```
First_id: INTEGER = 1000
```

Это объявление вводит целочисленный константный атрибут *First_id*. Существует следующее соглашение:

Почувствуй стиль

Константы
Имена константных атрибутов, как и имена предопределенных объектов, начинаются с заглавной буквы и продолжаются строчными буквами в нижнем регистре.

Этот стиль является общепринятым и для строк, как в следующем объявлении:

```
Map_title: STRING = "Plan of the metro"
```

Строка в правой части называется **манифестной строкой**.

Не будучи переменными, константные атрибуты не могут использоваться в качестве цели в операторах присваивания: *First_id:*= 2 или *Map_title:*= «Something else». Это примеры ошибочных присваиваний (можете проверить и увидеть сообщения компилятора).

Константные атрибуты дают имена значениям, создавая именованные константы. Следует систематически использовать такой способ работы с константами.

Почувствуй методологию

Принцип символических констант
Когда программе требуются константы, за исключением совсем простых случаев (например, констант 0 и 1), в операторах не следует использовать манифестные константы (константы без имени). Объявите константный атрибут с соответствующим значением и затем всюду используйте этот атрибут.

Если понадобилась строка для вывода сообщения об ошибке или понадобилась физическая константа, то не используйте ее непосредственно в операторе, как в следующем примере:

```
display ("Не могу послать сообщение в указанное время!")
length := 2.54 _ length_in_inches
```

Вместо этого объявите:

```
Timeout_message: STRING = "Не могу послать сообщение в указанное время!"
Inches_to_centimeters: REAL = 2.54
```

а затем используйте именованные константы в операторах:

```
display (Timeout_message)
length := Inches_to_centimeters * length_in_inches
```

Можно привести два аргумента в пользу этого правила.

- Читабельность: правило рекомендует давать каждой константе имя, объясняющее ее роль.
- Упрощение эволюции программ: значения констант могут изменяться в ходе развития системы. В этом случае легко найти и изменить объявление константы, не разыскивая в тексте все точки ее использования. Общепринято все важные константы, например, сообщения об ошибках, группировать в одном месте, помещая их в отдельный класс, спроектированный только для этой роли.

Непосредственное использование манифестных констант в операторах вообще не рекомендуется, это особенно верно для строк. Успешные системы часто имеют международные версии, поэтому система должна поддерживать пользовательский интерфейс на языке, выбранном пользователем системы. В этом случае фактические строки приходят из файла, входящего в ресурсы системы.

9.5. Ссылочное присваивание

Значения, которыми мы манипулируем, – в частности, поля объектов, соответствующие атрибутам их классов, – могут быть базисными значениями, такими как целые и булевские, а могут быть и ссылками. До сих пор мы применяли присваивание к базисным значениям, но нам понадобится и присваивание ссылок. Это необходимо, например, при построении связанных структур данных, таких как список остановок метро, где каждая остановка содержит ссылку на связанную с ней станцию и на следующую остановку на линии.

Построение остановок метро

Реализация класса *STOP* требует такого ссылочного присваивания. Интерфейс класса включает спецификации следующих компонентов:

```
set_station (ms: STATION)
        -- Связать эту остановку с ms
    require
        station_exists: ms /= Void
    ensure
        station_set: station = ms
link (s: STOP)
        -- Сделать s следующей остановкой на линии
    ensure
        next_set: right = s
```

Реализация требует установки значений для атрибутов класса *station* и *right*, для чего необходимо присваивание. Приведем текст метода (не ограничиваясь интерфейсом):

```
    set_station (ms: STATION)
            — Связать эту остановку с ms
        require
            station_exists: ms /= Void
        do
            station:=ms
        ensure
            _set: station = ms
        end
    link (s: STOP)
            — Сделать s следующей остановкой на линии.
        do
            right := s
        ensure
            next_set: right = s
        end
```

Ссылочное присваивание **присоединяет** ссылку к новому объекту. Предыдущее значение ссылки могло быть void (объект отсутствует) или присоединено к другому объекту (могло быть даже присоединено к тому же самому объекту, и тогда присваивание ничего бы не меняло). Для иллюстрации этих возможностей рассмотрим переменные *s1* и *s2* типа *STOP* и два оператора создания:

```
create s1.set_station (Station_Balard)
create s2.set_station (Station_Issy)
```

Оба оператора используют процедуру создания *set_station*. Это необходимо, поскольку мы уже переписали класс *STOP* следующим образом:

```
class STOP create
    set_station
feature
    station: STATION
    right: STOP
    set_station (s: STATION) … Как выше …
    link (s: STOP) … Как выше …
invariant
    station_exists: station /= Void
end
```

Операторы создают два объекта:

Благодаря процедуре создания *set_station* ссылки на станцию присоединяются к двум предопределенным объектам типа *STATION*. Ссылки *right* остаются void, так все ссылки изначально устанавливаются в void, а *set_station* ничего не делает с *right*.

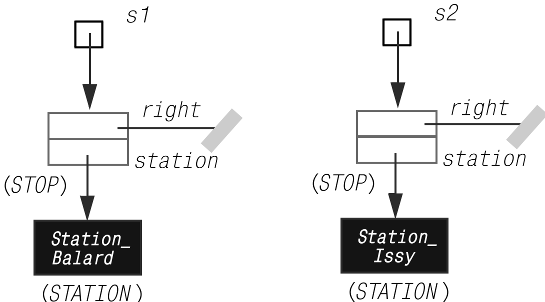


Рис. 9.9. Создание двух остановок

Построение линии метро

Для соединения двух остановок можно использовать оператор:

```
s1.link(s2)
```

Он обновляет *right*-ссылку первого объекта:

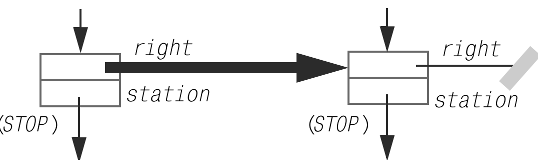


Рис. 9.10. Сцепление двух остановок

Результат является следствием ссылочного присваивания в процедуре *link*:

```
link (s: STOP)
        -- Сделать s следующей остановкой на линии.
    do
        right:=s
    ensure
        right_set: right = s
    end
```

Это пример ссылочного присваивания, присоединяющего ссылку. Здесь ссылка (поле *right* объекта *STOP* слева) была до присваивания равна void; мы присвоили ей непустую ссылку *s2*, в результате *right* была присоединена ко второму *STOP*-объекту. Ссылочное присваивание можно использовать для отсоединения от объекта, делая ссылку пустой – void. Для примера добавим в класс *STOP* следующую процедуру:

```
make_last
        -- Сделать эту остановку последней на линии.
```

```
do
      right:= Void
ensure
      no_right: right = Void
end
```

Здесь используется значение **Void**, всегда обозначающее пустую ссылку. Следующие три вызова дают один и тот же эффект в предположении, что *v* имеет значение void:

```
s1.make_last     s1.link (Void)      s1.link (v)
```

Чтобы еще немного поиграть со ссылками, рассмотрим предыдущий пример, но уже с тремя остановками:

```
create s1.set_station (Station_Balard)
create s2.set_station (Station_Issy)
create s3.set_station (Station_Montrouge)
s1.link(s2)
s2.link(s3)
s3.make_last
```

Результатом является дополнение предыдущего рисунка, показывающее мини-линию метро:

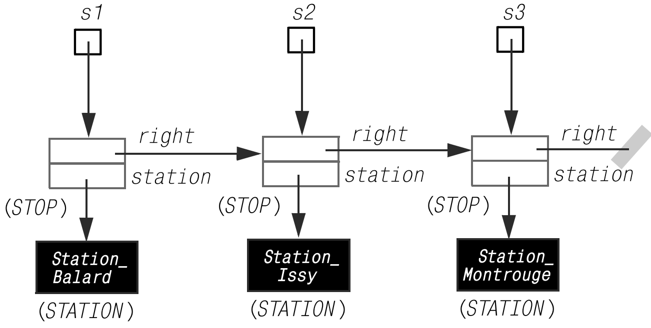


Рис. 9.11. Создание небольшой линии метро

9.6. Программирование со ссылками

Ссылка идентифицирует объект. Их использование дает ряд преимуществ: моделирование отношения «Знаю о», поддержка связанных структур. Но у ссылок есть и обратная, темная сторона: *динамические псевдонимы* заставляют быть крайне внимательными при работе со ссылками.

Ссылки как инструмент моделирования

Один объект может включать ссылку на другой объект, представляя концепцию «знаю о» (об этом объекте). Эту концепцию можно сравнить с концепцией «содержит» (другой объект). Разницу демонстрирует пример с двумя интерпретациями глагола «имеет».

- Автомобиль *имеет* марку (A car *has* a brand).
- Автомобиль *имеет* мотор (A car *has* an engine).

Ключевая разница в разделении: два автомобиля могут иметь одну и ту же марку, но ни один уважающий себя автомобиль не будет разделять мотор с другим автомобилем. Создавая ОО-программную модель, в первом случае используем ссылку на объект, во втором – подобъект.

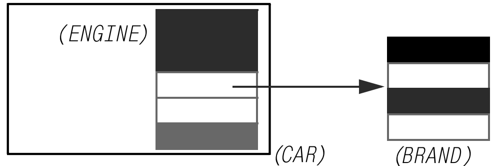


Рис. 9.12. Подобъекты и ссылки

Развернутые типы помогают моделировать подобъекты. Такая гибкость моделирования крайне важна при построении модели сложной системы.

Использование ссылок для моделирования связанных структур данных

Еще одно, уже продемонстрированное приложение ссылок состоит в представлении коллекции объектов, называемой также «контейнером» объектов. Коллекцию можно рассматривать как совокупность связанных ячеек, каждая из которых может содержать ссылки на другие ячейки. Примером может служить связный список, такой как наша линия метро, последовательная структура, где каждая ячейка, кроме последней, содержит ссылку *right* на следующий элемент.

Связные структуры облегчают выполнение таких операций, как вставка и удаление. Для примера удалим из мини-линии метро второй элемент, указав новую связь для *right*-ссылки. Эффект операции показан на рисунке:

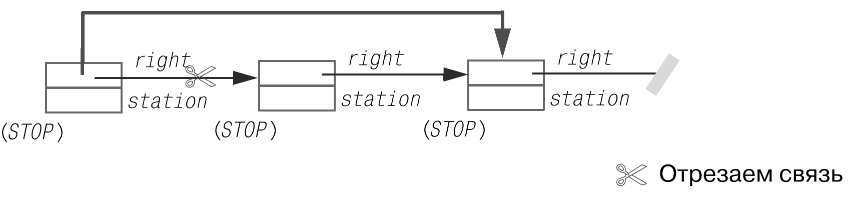


Рис. 9.13. Удаление ячейки из связной структуры

Рассмотрим возможную для класса *STOP*-процедуру, представляющую удаление следующей остановки на линии:

```
remove_right
        — Удаление следующей остановки.
    require
        not_last: right /= Void
    do
        right: = right.right
    ensure
        skipped_one: right = old right.right
    end
```

Альтернативой мог бы быть метод, в котором удалено предусловие, изменен заголовочный комментарий «— Удаление следующей остановки, если она имеется» и изменено тело процедуры следующим образом:

```
if right /= Void then right := right.right end
```

Аналогично, мы могли бы пожелать вставить еще одну остановку между текущей и следующей, если таковая имеется:

Рис. 9.14. Добавление ячейки в связную структуру

Вот соответствующий метод (предполагается, что он должен быть добавлен к классу *STOP*):

```
put_right (s: STOP)
        — Добавить к линии метро остановку s после текущей остановки,
        — оставляя любые последующие остановки.
    require
        exists: s /= Void
    do
        s.link (right)  — Операция 1
        right := s         — Операция 2
    ensure
        linked_to_new: right = s
        retained_others: right.right = old right
    end
```

Метод работает независимо от того, присоединена ли *right* к объекту или имеет значение void (текущая остановка является последней). Новая ячейка *s* не должна быть пустой. Предыдущее значение ее *right*-ссылки, каково оно ни было – пусто или присоединено, будет потеряно при выполнении метода *link*, но *station* станция, с ней связанная, останется.

Как и в алгоритме свопинга – обмена данными двух значений – порядок присваивания важен. Мы связываем *s* с ее новым соседом (операция 1 на рисунке), представленным полем *right*, в его исходном значении. Только после этого можно изменить значение *right*, применяя операцию 2.

Процедуры *remove_right* и *put_right* иллюстрируют общие схемы манипуляции со связанными структурами.

В большинстве практических случаев интерфейс будет слегка отличен. Операции вставки не будет передаваться в качестве аргумента элемент списка, такой как *STOP*-объект в нашем примере. Вместо этого аргументом будет объект типа *STATION*, после чего в методе будет создан элемент списка с присоединенной станцией и вновь созданный элемент будет вставлен в список. Мы будем изучать такие операции при обсуждении связных списков.

Void ссылки

Третье преимущество ссылок предоставляет значение **Void**, используемое, в частности, для завершения связанных структур, символически изображенное на последних рисунках.

Как вы знаете, этот благословенный дар опасен: возможность, что объект *v* имеет значение void на некотором шаге некоторого сеанса выполнения, усложняет программирование, требуя проверки цели каждого вызова *v.f* (...), поскольку необходимо гарантировать, что *v* никогда не будет void при любом выполнении вызова.

Это и есть та цена, которую приходится платить за гибкость описания связанных структур данных. Сопроводим эту ситуацию методологическим советом.

Почувствуй методологию

Использование void-ссылок
Резервируйте void-ссылки для завершения связанных структур.

Это означает, что не следует использовать void для представления специальных значений типов, не задающих связанные структуры. Например, создавая класс *ACCOUNT* в программе, моделирующей работу с банковскими счетами, не следует использовать void для представления ошибочного счета, лучше создать специальный объект – «Неизвестный_счет». Это избавит вас от риска вызова метода класса ссылкой со значением void. Конечно, необходимо позаботиться о разборе ситуации, когда встречается специальный объект, поскольку вы также не хотите, чтобы метод выполнялся, но давал неправильные результаты.

Обращение списка

Процедуры *remove_right* и *put_right* дают хороший пример работы со связанными структурами. Их простые, но уже не тривиальные алгоритмы демонстрируют заботу о void-значениях.

Для дальнейшего знакомства со ссылочными алгоритмами давайте рассмотрим более сложный в реализации алгоритм обращения списка. Чтобы не слишком усложнять задачу, будем рассматривать задачу создания нового списка, элементы которого связаны в обратном порядке по отношению к исходному списку:

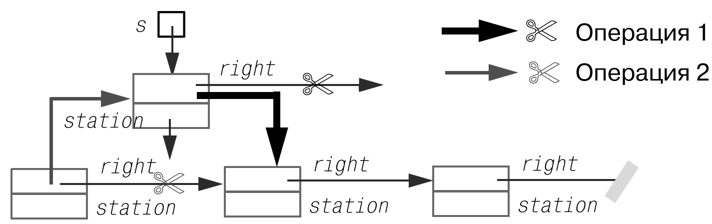


Рис. 9.14. Получение обращенной версии связного списка

Мы начинаем с *s* – ссылки на линию метро. Так как каждая остановка метро имеет ссылку на следующую остановку, можно, используя *s*, получить доступ ко всей линии, последовательно применяя *right*. Мы не хотим модифицировать эту структуру, но хотим создать новую, доступную через **Result** в нашей функции, которая будет содержать те же элементы, но сцепленные в обратном порядке. Для иллюстрации на рисунке показана информация, связанная с каждым *STOP*-объектом (станции, заданные также ссылками, представлены номерами от 1 до 5).

Всякий раз, когда предлагается некоторая задача, разумно перед дальнейшим чтением попытаться самому найти ее решение.

В таких алгоритмах важна производительность. Первый элемент нового списка является последним элементом исходного списка, поэтому для его получения нужно пройти весь исходный список. Для получения второго элемента снова нужно пройти исходный список до предпоследнего элемента. Это плохая стратегия, требующая порядка n^2 операций, если в списке *n* элементов. Вместо этого мы хотим выполнить обращение списка, проходя исходный список лишь один раз.

Как для любого итеративного алгоритма, ключом для нахождения правильного алгоритма или для понимания существующего алгоритма, является инвариант цикла, задающий свойства на каждом шаге цикла. Рисунок ниже показывает ситуацию, возникающую на одной из итераций пока еще не написанного цикла.

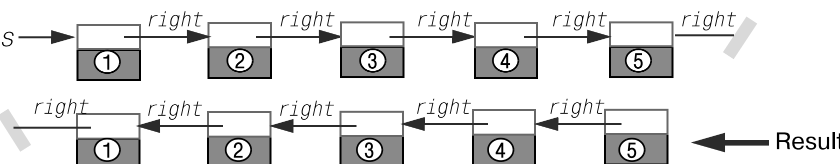


Рис. 9.15. Создание обращенной версии односвязного списка: промежуточное состояние

Вот что мы сделаем для получения обращенной формы части исходного списка. Введем две переменные – они будут локальными переменными метода, которые будут показывать, насколько мы продвинулись по исходному циклу:

- *previous* указывает на последнюю ячейку, уже включенную в обращенный список;
- *pivot* указывает на первую еще не обработанную ячейку, получая значение void, когда мы обработаем все ячейки исходного цикла. Так что условие *pivot* = **Void** будет сигнализировать, что мы все сделали, и может служить условием выхода из цикла.

Эти два свойства задают инвариант цикла. Схема простая. На каждой итерации цикла обрабатывается следующая ячейка, известная как *pivot*. Создается клон этой ячейки, который и добавляется в новый список, становясь его началом, что нетрудно сделать, зная **Result**, ссылку на начало списка. После этого в исходном списке передвигаются вправо *previous* и *pivot,* что восстанавливает истинность инварианта. Вот описание метода:

```
reversed (s: STOP): STOP
            -- Новая остановка - первая на новой линии, имеющей те же станции,
            -- что и s, но идущие в обратном порядке.
```

```
        -- Нет предусловия, поскольку метод работает и для s, представляющей
        -- пустой список.
    local
        previous, pivot: STOP
    do
        from
            previous := Void; pivot := s
        invariant
            -- Список с началом Result содержит все ячейки исходного списка
            -- от начала и до previous включительно; pivot задает следующую
            -- ячейку, если она есть.
        until
            pivot = Void
        loop
            Result := pivot.cloned; Result.link (previous)
            previous := pivot; pivot := pivot.right
        variant
            --  Смотри ниже.
        end
    end
```

Нам необходима локальная переменная *previous* для сохранения предыдущего значения *pivot* на момент создания новой ячейки. Для инициализации *previous* используется значение **Void**. Вызов функции *acloned* позволяет создать новый объект (аналогично оператору создания), дублирующий поле за полем объекта *a*.

В зависимости от используемой версии библиотеки для тех же целей может использоваться *twin*, старое имя *cloned*.

На рисунке показаны детали добавления ячейки.

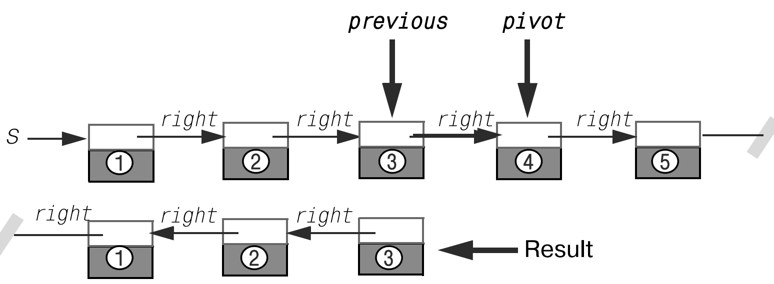


Рис. 9.16. Создание обращенной версии односвязного списка: добавление элемента

Следует убедиться, что алгоритм всегда применяет квалифицированные вызовы *pivot.cloned* и pivot.*right* в теле цикла с непустым *pivot*.

При изучении рекурсии появится интересное упражнение, требующее переписать метод *reversed* с использованием рекурсии вместо цикла.

Делаем списки явными

В последующих главах мы вернемся к связным спискам для более систематического их изучения. В частности, построим процедуру обращения списка на месте вместо функции, создающей новый список. Понятно, что такая процедура более сложна, так как требует перестройки начальной части существующего списка, не затрагивая его остатка.

> Поскольку никакие новые механизмы при этом не вводятся, то уже сейчас вы можете попытаться написать эту процедуру.

Это даст нам возможность ввести более общие рамки для связанных структур, чем те, что рассматривались в последних примерах. Мы уже работали с классом *STOP*, чьи экземпляры содержат ссылку *right* на следующий экземпляр:

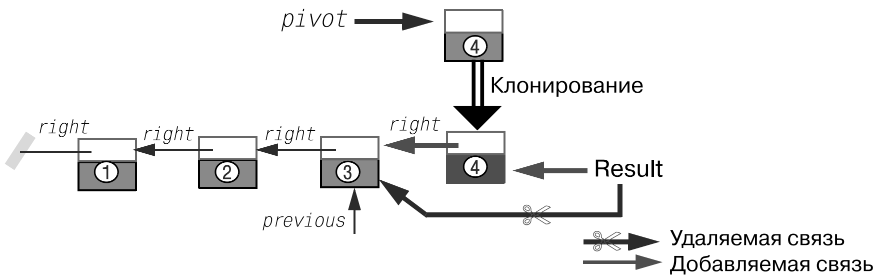


Рис. 9.17. Неявная структура списка

Доступ к остановке *s*, благодаря последовательному применению *right*, дает нам доступ ко всем последующим элементам, но сама структура списка остается неявной. Как следствие, сложно выразить вариант цикла в вышеприведенном примере, в то время как он интуитивно ясен – на шаге *i* мы выполнили обращение *i* элементов исходного цикла. Но у нас нет возможности ссылаться на такие глобальные свойства, поскольку мы имеем дело только с отдельными элементами. При систематическом подходе список сам является объектом:

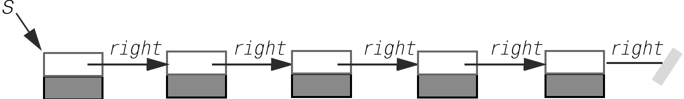


Рис. 9.18. Связный список

Объект «список», показанный вверху, является экземпляром *LINE*. Он используется только как заголовок списка, не содержащий его элементов (эту роль играют экземпляры *STOP*), но он хранит общую информацию о списке: число элементов – *count*, ссылку *first_element* на первый *STOP*, позволяющий далее добраться до всех других элементов.

> Для реализации понятия списка с курсором, часто используемого для облегчения итерирования, достаточно включить в заголовок списка еще одну ссылку, указывающую на текущую позицию курсора.

В библиотеке EiffelBase ячейки списка (такие как экземпляры *STOP*) принадлежат библиотечным классам, таким как *LINKABLE*, в то время как заголовок списка принадлежит классу, такому как *LINKED_LIST*.

Полезно выполнить следующее упражнение – переписать *remove_right*, *put_right* и *reversed* как методы класса *LINE*, их естественное место в этом классе, а не в классе *STOP*. При этом необходимо тщательно следить за void-значениями и граничными случаями (пустые линии, вставка и удаление на концах списка).

Где используются операции над ссылками?

Мы упростили задачу, работая только с ячейками списка, а не со списком и ячейками одновременно, и не стали выполнять реверс списка на том же месте. Тем не менее, приведенные примеры являются хорошими образцами для понимания ссылочного программирования.

Понятно, что это работа, требующая аккуратности. Необходимо учитывать все возможные случаи, пустые или почти пустые структуры, следить за состоянием курсора, если он используется, обращать особое внимание на возможность void-ссылок, чтобы они не стали целью вызова. Серьезную проблему, которую мы вскоре начнем рассматривать, представляет появление «динамических псевдонимов», крайне затрудняющих доказательство свойств программ.

Чтобы стать исследователем или специалистом по инженерии программ, необходимо полное владение соответствующими приемами работы. В этой книге еще появятся некоторые алгоритмически сложные задачи, требующие ссылочного программирования. Надеюсь, они помогут в овладении нужными приемами.

Разрабатывая архитектуру системы, помните, что ссылочное программирование должно выноситься в отдельные классы, а не включаться непосредственно в модули, моделирующие проблемную область.

Почувствуй методологию

Принцип ссылочного программирования

Нетривиальные действия со ссылками, такие как операции вставки и удаления элементов связанных структур, не должны появляться в проблемно ориентированных частях программы. Для них должны проектироваться специальные библиотечные классы. Если по каким-либо причинам библиотеки классов недоступны, то из классов следует создавать специальные кластеры.

«*Проблемно ориентированные части программы*» – это те компоненты, которые непосредственно отвечают за достижение целей, стоящих перед программой: обработку телефонных звонков, управление продажами, форматирование текста...

Здесь редко встречаются прямые действия со ссылками, хотя они могут быть крайне полезны в *реализации* концепций проблемной области. Система, обрабатывающая текст, может быть организована в виде списка абзацев. Но если в списке абзацев нет ничего специфического, что отличало бы его от списка произвольных элементов, то обработка должна выполняться в общих классах, созданных для таких структур.

Если вам самим придется писать реализацию, то принцип ссылочного программирования потребует создания специального кластера, отделяя «поддерживающую технологию» от чисто проблемных задач.

К счастью, современные среды разработки обеспечивают библиотеки компонент, включающие базисные типы структур объектов. Таковой является библиотека EiffelBase, разраба-

тываемая многими людьми в течение многих лет. Поддерживающие библиотеки есть и у языков Java, C#, C++:

Почувствуй методологию

Принцип библиотечных фундаментальных структур данных
Для фундаментальных структур данных и алгоритмов используйте компоненты базисных библиотек, если они применимы, а не разрабатывайте собственную реализацию.

Если нужен список, дерево, стек или очередь, любая фундаментальная структура данных – первым делом проверьте, нет ли ее в библиотеке, соответствует ли она вашим потребностям. И если да, то не программируйте сами, а используйте библиотеку.

Этот совет применим не только к действиям над ссылками: зачем повторно разрабатывать, когда можно повторно использовать?

Даже если вы должны написать собственную реализацию, отличающуюся от библиотечной, библиотека может помочь вам. Прежде чем все писать с чистого листа, можно начать с существующего класса и модифицировать его. Помните, однако, общий совет о дублировании кода. Прием «copy – paste» считается ужасным при разработке ПО.

Наследование часто предлагает вам взять один или несколько существующих классов и построить новый класс, расширяющий, переопределяющий, адаптирующий их возможности, не изменяя родителей и не копируя их код. Эта интересная форма повторного использования характерна для ОО-подхода, она отличается от используемой до сих пор формы клиентского класса, более тонкая, в то же время более мощная и часто более полезная.

Динамические псевдонимы

Давайте вернемся немного назад и посмотрим, что же делает ссылки коварными (этот раздел можно считать дополнительным и пропустить при первом чтении).

В дополнение к проблеме пустоты ссылок у них есть еще одна особенность – они позволяют одному объекту иметь несколько имен. Такие имена называются псевдонимами, как в случае, когда у некоторых людей, чаще писателей, есть псевдонимы (Марк Твен – это псевдоним писателя Самюэля Клеменса). Ссылочное присваивание приводит к появлению динамических псевдонимов, определяемых в период выполнения. Рассмотрим присваивание:

```
b := a
```

Оно является причиной появления динамических псевдонимов, поскольку в результате присваивания *a* и *b* присоединены к одному и тому же объекту, являются его именами. Мы не можем полагать, что так будет всегда, поскольку это может встретиться при одном выполнении программы и не проявиться в других сеансах, – все зависит от структуры управления.

Почему динамические псевдонимы опасны? Проблема в том, что они создают дополнительные трудности в доказательстве свойств программы. Предположим, что можно выразить свойство объекта, используя его имя *b*. Пусть теперь выполняется операция над объектом, используя имя *a*. Операция может повлиять на свойство *b*, но как учесть это влияние, если операция вообще не упоминает имя *b?*

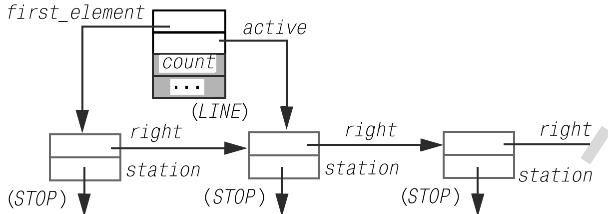


Рис. 9.19. Псевдонимы как результат ссылочного присваивания

Рассмотрим следующую схему:

```
– Здесь вы знаете, что для b выполняется свойство: P (b)                    [5]
OP (a)        -- Операция, выполняемая над a и не упоминающая b
– Можно ли быть уверенным, что P (b) по-прежнему будет выполняться?
```

Ответ на последний вопрос будет утвердительным, если *a* и *b* являются различными переменными базисных типов, таких как *INTEGER*: ничего из того, что вы делаете с *a,* не влияет на *b*.

Со ссылками и динамическими псевдонимами все становится по-другому! Предположим, что речь идет об объектах класса *STUDENT* и процедура *raise* увеличивает рейтинг.

Рассмотрим схему:

```
– Здесь вы знаете, что b.gpa меньше 4
a.raise
– Что можно сказать теперь о b.gpa?
```

Эта схема может встретиться в методе с явным контрактом:

```
dubious
        – Иллюстрация опасностей псевдонимов.
    require
        low_gpa: b.gpa < 4
    do
        … Возможно, другие инструкции …
        a.raise
    ensure
        – Что можно утверждать о b.gpa?
    end
```

Ответ на последний вопрос зависит от того, является ли *b* псевдонимом *a*. Если нет, то выполнение метода сохранит *b.gpa* и не изменит все другие свойства *b*. Но, если *b* – псевдоним *a*, то свойство, установленное в предусловии, нарушится и не может служить частью постусловия. Если условие *gpa* < 4 является частью инварианта класса, то, будет ли *dubious* сохранять его, также зависит от наличия псевдонимов.

Проблема не в семантической неопределенности. Нет никаких сомнений, что все, что происходит, определяется тем, являются ли *a* и *b* псевдонимами. Трудности возникают в на-

шем методе вывода утверждений о поведении программы. При рассмотрении семантики оператора мы должны учитывать возможность существования псевдонимов. Это предохранит нас от ложных заключений, что если в операторе не встречается имя переменной, то оператор не может повлиять на любое свойство, связанное с этой переменной.

Действительной трудностью является то, что псевдонимы не знают границ, они проникают сквозь границы модулей вашей программы и потенциально могут распространяться на всю программу. Если бы речь шла только о присваивании в пределах одного класса, как в нашем примере, то с этим легче было бы справиться, зная область действия псевдонимов. Но передача аргументов методу *r* (*a: T*), в момент вызова *r* (*b*) приводит к появлению динамических псевдонимов, совершенно аналогично присваиванию переменных. Так что, как только вы передаете ссылку методу, который может передать ее далее другому методу, процесс распространения псевдонимов может повлиять на удаленные части всей программы и воздействовать на объект, который, как вы наивно предполагаете, принадлежит лично вам.

Возможность удаленной модификации часто желательна. Проблема в том, чтобы точно знать, как все происходит. Можно заметить, что сложности связаны главным образом со спецификациями. Предыдущая процедура может быть переписана так, чтобы постусловие более точно отражало наши намерения:

```
dubious
        — Иллюстрация опасностей псевдонимов.
    require
        low_gpa: b.gpa < 4
    do
        … Возможно, другие инструкции …
        a.raise
    ensure
        a.gpa = old a.gpa + Increment
    end
```

Здесь *Increment* – это значение, добавляемое в методе *raise*. Как и ожидалось, выполнение влияет на a. Но эта спецификация ничего не говорит о том, что не должно происходить, – что другие объекты, такие как *b,* не должны изменяться. Конечно, можно было бы добавить кучу предложений постусловия в форме:

```
        b = old b
        c = old c
        … и так для каждого атрибута класса …
```

Все это ужасно и мало привлекательно.

Вопрос в том, как написать практический контракт, который бы не только специфицировал, как изменяются некоторые свойства, но перечислял бы все свойства, которые не должны изменяться. Последние известные как **рамочные свойства**, а в целом проблема называется **проблемой рамок**.

Эта проблема ныне находится на этапе исследований. Пока еще не найден консенсус для выражения рамочных свойств, в отличие от принципиального соглашения по представлению (отличающегося деталями) обычных контрактов – предусловий, постусловий, инвариантов.

Обсуждение проблемы рамок далеко выходит за пределы нашего рассмотрения. Но следует понимать, что из-за динамических псевдонимов возникают серьезные проблемы с выводом утверждений о корректности работы программы. Практически это означает, что следует быть крайне внимательным при работе со ссылками, особенно при передаче объектов другим методам. По этой причине, как обсуждалось, нетривиальные операции над ссылками лучше возлагать на специализированные библиотеки и кластеры. Теперь вы знаете более точно, что понимается под «нетривиальностью»: любые действия, которые могут стать причиной появления псевдонимов и приводить к неожиданным эффектам.

Еще одно общее замечание. Можно попытаться всю вину возложить на ссылки и предположить, что механизм неудачен и от него следует отказаться. Но эффект от этого, видимо, был бы еще хуже, поскольку дело не в ссылках. Старейший язык программирования Фортран не имел ссылок, но разрешал помещать данные в общий пул (массив), доступный для всех модулей. Элементы данных были доступны по индексу, что аналогично псевдонимам. Риски те же, но еще страшнее, поскольку работа ведется на более низком уровне абстракции.

Псевдонимы не являются программистской концепцией. Это свойство человеческой цивилизации – именовать вещи и давать одной и той же сущности разные имена. «*Прекрасная дочь Леды, супруга несчастного Менелая, любовница Париса*» – все это ссылки на одну и ту же героиню – прекрасную Елену Троянскую. Филологи вводят различные термины для такой синонимии: метафора, метонимия, синекдоха, аллегория.

Вот пример из обыденной жизни. Ваш друг, говоря об общей знакомой студентке, сообщил: «Мария озабочена, поскольку перед экзаменом ее рейтинг составлял 3,7». Вскоре вы услышали разговор студентов: «Моя подруга прекрасно сдала экзамен, но ее рейтинг может возрасти максимум на 0,4». Можете ли вы полагать, что в разговоре речь шла о Марии? Сомнительно, в псевдонимах нельзя быть уверенными. Возможно, у Марии все прекрасно, но гарантировать это нельзя.

Так что не стоит винить программистов в нечеткости мышления. Они, как и все, радуются и страдают от софизмов, свойственных образцам интеллектуального вывода.

9.7. Ключевые концепции, изучаемые в этой главе

- Программам требуется в ходе выполнения изменять значения своих переменных (исключение – программы, написанные на функциональных языках). Основным способом является присваивание.
- Присваивание применяется как к базисным значениям, которые копируются, так и к ссылкам, для которых копируется только ссылка, но не связанный с ней объект.
- Ссылочное присваивание вводит динамические псевдонимы, в результате которого объект становится доступен через различные имена. Это усложняет рассуждения о программе.
- Переменные включают атрибуты, представляющие поля объекта, и локальные переменные, используемые только внутри отдельных методов.
- Ссылки позволяют описать и выполнять действия над связанными структурами данных. Примером операции может служить обращение структуры данных.

Новый словарь (словарь 1)

Assignment	Присваивание	**Attribute**	Атрибут
Local variable	Локальная переменная	**Temporary variable**	Временная переменная
Variable entity	Переменная сущность		
Variable	Переменная		

Точная терминология атрибутов и методов – feature (словарь 2)

Attribute	Атрибут	**Command**	Команда
Feature	Компонент класса Метод или атрибут (в зависимости от контекста)	**Procedure**	Процедура
		Query	Запрос
		Function	Функция

9-У. Упражнения

9-У.1. Словарь

Дайте точные определения терминам словаря (словарь 1).

9-У.2. Словарь

Дайте точные определения терминам словаря (словарь 2).

9-У.3. Концептуальная карта

Добавьте новые термины в концептуальную карту, построенную в предыдущих главах

9-У.4. Терминология

1. Является ли каждая функция сущностью?
2. Является ли каждая сущность функцией?
3. Может ли функция быть запросом?
4. Является ли **Result** атрибутом?
5. Является ли **Result** функцией?
6. Является ли **Result** сущностью?
7. Является ли **Result** переменной?
8. Все ли переменные локальны?
9. Является ли каждый атрибут сущностью?
10. Является ли каждый метод запросом?
11. Является ли каждый запрос сущностью?
12. Является ли каждый атрибут переменной?
13. Является ли каждая функция переменной?
14. Является ли каждая сущность переменной?
15. Может ли запрос быть переменной?

16. Может ли функция быть переменной?
17. Является ли каждая переменная сущностью?

9-У.5. Обмен значениями (свопинг)

Пусть *var1* и *var2* – переменные типа *INTEGER* с обычными арифметическими операциями. Можете ли вы написать операторы, выполняющие обмен значениями без использования любых локальных переменных и иных сущностей? (Ответом является старый программистский трюк) Какие ограничения при этом накладываются?

9-У.6. Процедура обращения

Пусть атрибут *s* в классе C обозначает ссылку на первый *STOP*-объект линии метро. Напишите процедуру *reverse*, обращающую порядок остановок на линии, так, чтобы *s* задавал теперь первую остановку в модифицированной линии (последнюю в оригинале). У процедуры *reverse* нет аргументов (если в процедуре задать аргумент *s*, это усложнило бы задачу, так как нельзя присваивать значение формальному аргументу, хотя и можно модифицировать присоединенный объект). Подсказка: используйте функцию *reversed* в качестве источника вдохновения.

9-У.7. Изменение остановок двумя способами

Внесите изменения в класс *STOP*, такие, чтобы каждая остановка была связана как с правым, так и с левым соседом, если он есть. Например, *link* метод должен теперь вызывать *link_right* и *link_left*.

1. Добавьте *put_left* в дополнение к *put_right* и *remove_left* в дополнение к *remove_right*.
2. Обновите *reversed*.

9-У.8. Список остановок как класс

Если вы выполнили предыдущее упражнение, то перепишите процедуры *remove_right, put_right, reversed,* а также *reverse* как методы класса *LINE*, а не *STOP*.

9-У.9. Правила языка

В присваивании *var:= exp,* левая часть *var* должна быть переменной, она не может быть выражением, включающим квалифицированный вызов, такой как *some_object.one_of_its_fields*. В чем причина появления такого правила? (Подсказка: обратитесь к принципу скрытия информации и инвариантам класса)

Часть II

Как все устроено

Во второй части сделаем перерыв и немного отдохнем от проектирования ПО и методов реализации. Займемся исследованием некоторых поддерживающих технологий:

- **компьютерами (пойдем немного дальше первоначального рассмотрения);**
- **описанием синтаксиса языков программирования и другими формальными нотациями;**
- **разнообразием языков программирования;**
- **инструментарием, поддерживающим разработку ПО.**

10

Немного об аппаратуре

Без компьютера ПО работать не может. Для понимания того, как разработать *хорошую* программу, необходимо понимать, как устроено «железо», на котором эта программа работает. В этой главе рассмотрим необходимые нам основы устройства аппаратуры, детализируя элементы рисунка, ранее уже показанного:

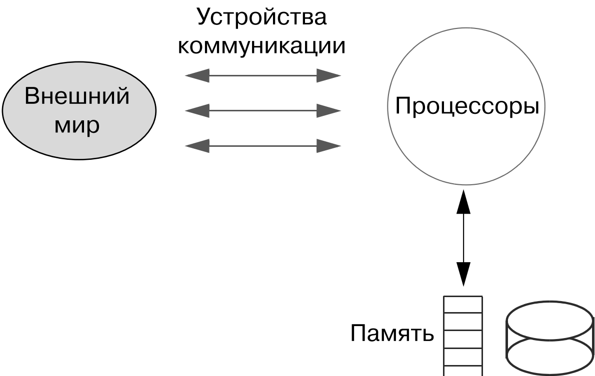


Рис. 10.1. Компоненты компьютерной системы

Начнем с рассмотрения феномена роста характеристик компьютера: объема информации, представленной данными компьютера, скорости доступа к этим данным, скорости выполняемых операций над данными.

Мы сознательно ограничимся лишь теми свойствами, которые необходимы программистам и важны для понимания тем, изучаемых в этой книге. В какой-то момент вы наверняка

прослушаете курс, такой как «Введение в архитектуру компьютеров», который даст вам более глубокие знания по устройству компьютеров.

10.1. Кодирование данных

Данные, хранимые в памяти компьютера, представляют информацию самой разной природы: анкеты работников, фотографии, музыкальные произведения, книги с их разнообразием форматирования, ну и, наконец, численные данные, результаты научных расчетов. Не забудем и о программах, хранимых в памяти. Необходим некоторый общий прием, позволяющий сохранять информацию и должным образом интерпретировать соответствующие данные.

Двоичная система счисления

Двоичная система счисления сделала возможным компьютерную революцию, поскольку открыла простой и общий способ представления информации.

В основе двоичной системы лежат два значения (отсюда «двоичная»). Сами значения не несут особого смысла и их можно называть по-разному: Белые и Черные, Чук и Гек или Изида и Озирис. Имеет значение лишь то, что они различны. Фактически мы будем называть их 0 и 1.

Термин «бит» означает математическую переменную, чьи возможные значения как раз и есть 0 и 1. Он придуман инженерами в конце 1940-го года как сокращение двух слов «**bi**nary digi**t**», подчеркивающее, что бит подобен цифрам обычной арифметики (0, 1, ... 9), но только с двумя возможными цифрами.

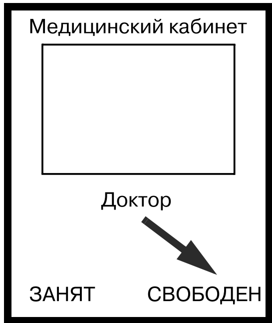


Рис. 10.2. Бит (техническая версия)

Бит также обозначает техническое устройство с двумя возможными состояниями, следовательно, способное быть представленным математическим битом, стоит только договориться, что есть 0, а что 1. Флажок на кабинете доктора, который может находиться в одном из состояний: «доктор свободен» и «доктор занят» – является битом. Для компьютерной индустрии важную роль играют «электронные» биты, где два состояния соответствуют двум разным напряжениям или намагниченным и размагниченным участкам, как на магнитной ленте или диске.

Причина, по которой двоичная система так хорошо себя зарекомендовала в том, что электроника сделала возможным:

- реализовать физические биты и упаковать их в небольшом объеме – если быть совсем точными, то очень много битов в очень маленькой области. Ниже мы увидим некоторые примеры;
- быстро писать и читать биты. Очень быстро;
- дешево создавать множество таких совокупностей битов. Очень дешево.

Эти свойства обеспечили успех двоичной системе. Некоторые первые компьютеры использовали десятичную систему. Тогда компьютеры рассматривались как машины для вычислений, и казалось естественным для вычислений применять привычную для людей десятичную систему, пришедшую еще с тех давних времен, когда для счета использовались пальцы рук, а пальцев было 10, а не 8, и не 16 (само слово digit – «цифра» – произошло от латинского слова finger – «палец»). Но для компьютеров, построенных на электронике, двоичная система давным-давно вытеснила всех своих соперниц.

Как это касается нас, программистов? В большей степени, чем вы могли подумать. Действительно, программы мы пишем на приятном для нас языке программирования, где попрежнему используем привычную нотацию для записи чисел, например, 3.1415926524. Но, как только приходится рассматривать, как данные хранятся в памяти, сразу же приходится учитывать, что только двоичная система является родной для компьютеров, даже если запись числа выглядит непривычной для людей. Давайте познакомимся с некоторыми свойствами двоичной системы

Основы двоичной системы

Если информация, с которой вам приходится иметь дело, – нечто большее, чем результаты игры в «орёл или решка», то двух значений явно маловато. Базисные комбинации, которыми кодируются конечное множество данных любого размера, являются:

- **байт** – последовательность из 8 битов;
- **слово (word)** на новых компьютерах предполагает последовательность из 8 байтов или *шестидесяти четырех* битов. В прошлые два десятилетия слово обычно означало *тридцать два* бита, отсюда «64-битная архитектура» и «32-битная архитектура» компьютеров.

Ранее определение «слова» не было стандартизовано, и компьютеры использовали слова различной длины. До сих пор можно столкнуться с «отклонениями», но они редки. Байты всегда имели 8 битов и иногда называются *октетами*.

Сколько различных значений может задавать последовательность битов? Один бит позволяет задавать два значения: 0 и 1. С двумя битами возможностей становится уже четыре:

0	0
0	1
1	0
1	1

В общем случае, последовательность из n битов для любого целого $n > 0$ задает 2^n значений.

Базисные представления и адреса

Для базисных единиц:

- байт с его 8-ю битами имеет 256 (2^8) различных значений;

- слово из 32 битов имеет 2^{32} возможных значений, примерно 4 миллиарда, точное значение приведено в нижеследующей таблице.

Если, например, мы хотим хранить текст, то можно использовать один байт для хранения каждого символа текста. Может показаться, что 256 различных символов избыточно много для представления текста, но фактически это как раз то, что нужно, поскольку помимо 26 строчных и 26 заглавных букв латиницы необходимы цифры, специальные символы клавиатуры компьютера (такие как ~, !, @ и другие), акцентированные символы для букв западных языков (é, Ä) и так далее. Стандартное кодирование всех этих символов 8-битовым представлением известно как расширенный код ASCII (American Standard Code for Information Interchange). Оригинальный код ASCII использовал только 7 битов (128 значений) и не поддерживал акцентированные буквы.

Расширенный ASCII код имеет несколько вариантов, наиболее популярный известен как стандарт ISO 8859-1, покрывающий символы наиболее распространенных европейских языков.

Для кириллицы или иероглифов китайского языка расширенного ASCII недостаточно. Стандартом является код Unicode, использующий 4 байта для представления одного символа, что вполне достаточно для всех наиболее важных письменных языков.

Для представления символьных сущностей в Eiffel можно использовать тип *CHARACTER_8* для расширенного ASCII или *CHARACTER_32* для Unicode. Общим решением является применение типа *CHARACTER*, который приводится к тому или другому типу в зависимости от конфигурационных установок. С этим мы встретимся в примерах, включающих символьные типы.

Для численной информации общеупотребительной практикой является использование слова для хранения целых значений. Математическое множество целых бесконечно, но память компьютера способна хранить только конечное множество. Если использовать для хранения целых чисел 32-битное слово, оно позволяет задавать примерно два миллиарда отрицательных целых и два миллиарда положительных целых чисел. Для многих приложений этого достаточно. С 64-мя битами границы существенно расширяются. В наших программах тип для целых именуется *INTEGER*. Целые характеризует и тип *CHARACTER*. В установках конфигурации целочисленный тип может быть определен как *INTEGER_32* или *INTEGER_64*. Доступны в программах и типы *INTEGER_8* и *INTEGER_16*. Если приходится иметь дело только с неотрицательными целыми, то можно использовать тип *NATURAL* с его вариантами от *NATURAL_8* до *NATURAL_64*.

Для представления чисел, отличных от целых, – рациональных, таких как 3/2, или иррациональных (вещественных), таких как π, используются типы *REAL_32*, *REAL_64*. Опять-таки можно использовать общий настраиваемый тип *REAL*. В отличие от целых чисел для представления вещественных значений обычно 2^{32} значений недостаточно, поэтому чаще для представления вещественных чисел задействуется тип *REAL_64*.

Адресом элемента данных называется его позиция в пронумерованной памяти компьютера. Примеры типов данных: *CHARACTER_8*, *INTEGER_32* и *REAL_64* показывают, что элементы данных могут быть различных размеров (1, 4 или 8 байтов). Для обеспечения унификации в адресации памяти за единицу памяти принимается байт, а начальный адрес равен нулю. Так, если память начинается с размещения 1000 значений типа *INTEGER_64* на компьютере с 8-байтными словами, то первый свободный элемент памяти имеет адрес 8000.

Степени двойки

Уже по одной той причине, что n битов могут хранить 2^n значений, степени двойки важны для двоичной системы. Ниже перечислены некоторые элементы этой последовательности.

Программистам следует помнить первые 10 элементов, порядок и величину других, а также используемые аббревиатуры (кило и другие).

От вишен к байтам

В обычном, десятичном способе счета, аббревиатура «кило» представляет степень 10, точнее, 10^3, которая служит естественной мерой вещей. На рынке мы покупаем один килограмм вишен, равный тысяче – 10^3 – грамм, за один миллион – 10^6 – долларов вы едва ли купите приличный дом в Южной Калифорнии, один миллиард – 10^9 – долларов может продлить на несколько часов существование банка, падающего во время кризиса.

n	2^n	Аппроксимация степенями 10	Общепринятое имя (аббревиатура)	Официальное название (аббревиатура)
0	0			
1	1			
2	4			
3	8			
4	16			
5	32			
6	64			
7	128			
8	256			
9	512			
10	1024	10^3(тысяча)	Kilo (K) Кило	Kibi (Ki)
16	65536			
20	1 048 576	10^6 (миллион)	Mega (M) Мега	Mebi (Mi)
30	1 073 741 824	10^9 (миллиард)	Giga (G) Гига	Gibi (Gi)
32	4 294 967 296	4×10^9 (4 миллиарда)		
40	1 099 511 627 776	10^{12}(триллион)	Tera (T) Тера	Tebi (Ti)
50	1 125 899 906 842 624	10^{15}	Peta (P) Пета	Pebi (Pi)
64	18 446 744 073 709 551 616	1.8×10^{19}		

Эти единицы применимы и к другим измерениям, связанным с компьютерами, не только по отношению к памяти.

- Обмен данными по линии может проходить со скоростью в 1 Mbps (Megabit per second) – один Мегабит за секунду.
- Центральный процессор может работать со скоростью в 1 GHz (Gigaherz – Гигагерц), выполняя один миллиард базисных операций процессора за секунду. «Герц» – термин, заимствованный у физики, единица измерения частоты.

В то время как размеры памяти и адресация выражается в байтах, скорость передачи данных обычно представлена в «битах за секунду», или **bps**. Так что «56K модем», если он работает на предельной для него скорости, что практически не случается, может передавать 56000 бит в секунду.

Компьютерные инженеры предпочитают использовать степени двойки для выражения размеров памяти. Здесь и начинается путаница. Точнее, она началась, когда некто (имя его не сохранилось в истории) заметил, что два в десятой степени равно 1024, что примерно равно 10^3 – тысяче, и как следствие принял блестящее решение использовать десятичные аббревиатуры – кило для почти-тысячи (2^{10}), мега для почти-миллиона (2^{20}), гига для почти-миллиарда (2^{30}). С ростом степеней аппроксимация становится все хуже, как видно из таблицы.

Двоичная интерпретация тысячи наряду с традиционной иногда приводит к неразберихе, особенно если обе интерпретации применяются совместно. На вышедших уже из употребления флоппи-дисках их емкость указывалась как 1.44 MB, но означало это 1440 (десятичная интерпретация) раз по 1024 (двоичная интерпретация) байтов.

Чтобы покончить с беспорядком, соответствующая организация, занимающаяся стандартами, предписала применять десятичные аббревиатуры – кило и другие – для степеней 10, а для степеней двойки применять другие аббревиатуры – киби, меби, гиби, приведенные в последнем столбце таблицы. Эти имена не получили широкого распространения.

Откровенно говоря, в 2009 году их никто не использовал. Поиск в Google показал, что все ссылки на Gibibyte (Гибибайт) давали только определение термина, но не давали примеров его практического использования.

Во избежание недоразумений помните, что двоичная интерпретация применяется только для измерения памяти, в остальных случаях применяется десятичная. Так что 1-GHz компьютер с памятью в 1 GB выполняет один миллиард операций в секунду, но памяти имеет больше чем один миллиард, – на 73 миллиона байтов больше. На практике разница не столь уж велика: что для нас десяток-другой миллионов?

Действия над числами

Представление целых чисел в компьютере хорошо тем, что оно точно. Действия над целыми выполняются точно так же, как это принято в математике. Плохо лишь то, что целые в компьютере составляют конечное подмножество, в отличие от математики. Для 64-битного компьютера множество целых определяется диапазоном: {- 2^{63}, $+2^{63}$ -1}.

Точные значения границ диапазона зависят от представления отрицательных чисел в памяти компьютера. Обычно отрицательные числа хранятся в дополнительном коде. Тогда, если для хранения целого числа отводится N битов, то нижняя граница равна (-2^{N-1}), а верхняя ($+2^{N-1}$ -1), так что максимальное по модулю отрицательное число на единицу больше максимального положительного числа. Дополнительный код строится так, чтобы поразрядное сложение n и –n в двоичной системе давало ноль. Для положительных целых старший разряд в представлении числа N битами является знаковым и равен нулю, так что на само число остается N-1 бит, что и дает верхнюю границу. Отрицательные числа строятся добавлением единицы к обращению положительного числа (под обращением понимается взаимная замена 0 и 1). При N = 4 число 5 будет храниться как 0101, а число -5 – как 1011.

Для допустимых значений целых точным является не только их представление – результаты операций над целыми дают те же значения, что и их математические двойники, за исключением тех случаев, когда значения результатов выходят за пределы, допустимые для целых. Написав $a + b$ для целых a и b, получим корректный результат, если он не больше мак-

симального или не меньше минимального значения. Выход результата за допустимые пределы известен как «арифметическое переполнение».

Представление вещественных чисел – *REAL*-тип, как он называется в Eiffel (float – в других языках), аналогично представлению целых в том отношении, что задает конечное множество возможных значений. Как следствие, представление вещественных чисел частично и не всегда точно. В математике между любыми двумя вещественными числами находится бесконечно много других вещественных чисел. Это же справедливо и для рациональных чисел – менее мощного подмножества вещественных чисел, так что даже все рациональные числа не могут быть точно представлены в памяти компьютера. В этом отражается типичная разница между информацией и хранимыми данными.

> Стандартное представление вещественного числа состоит из трех частей: бита *s*, задающего знак числа, целого *n* – порядка числа, вещественного *f*, называемого нормализованной мантиссой, которая задает дробную часть, чья старшая цифра отлична от нуля. Вещественное число имеет вид $f \times 2^n$, со знаком *s*.

Эти свойства отражаются в программе тремя способами.

- Когда вещественное число *x* задается в программе явно программным текстом, или читается из файла, или поступает от датчиков или других устройств, то хранимое его значение является представимым значением вещественного типа, близкого к значению *x*, но не гарантированно с ним совпадающего.
- Арифметические операции – сложение, умножение, вычитание, деление, возведение в степень – могут быть невыполнимыми, хотя математически их результат строго определен. **Переполнение** (**Overflow**), упоминаемое выше для целых, здесь также возможно. Если *x* и *y* – два представимых вещественных числа (предположим для определенности, что они положительны), то $x + y$ или $x \times y$ могут быть слишком большими для представления. Еще один пример, характерный уже только для вещественных чисел: деление x / y, где *y* не равно нулю, но мало, так что результат в математике определен, но слишком велик для преставления в системе компьютера. Для вещественных чисел возможно возникновение ситуации, называемой «**Переполнение снизу**» (**underflow**), которая может возникать, например, при делении x / y, когда *y* слишком велико и результат слишком мал по абсолютной величине, чтобы он мог быть представлен значением, отличным от нуля (в этом случае говорят, что результат является «машинным» нулем).
- Даже в отсутствие переполнений сверху и снизу арифметические операции могут стать причиной возникновения ошибок. Если *x'* и *y'* являются представимыми значениями *x* и *y*, то программистская нотация $x + y$ не обозначает математическую нотацию суммы *x* и *y*, она даже не может представлять точное значение суммы $x' + y'$, поскольку не гарантируется представимость результата. Любой алгоритм, имеющий дело с вещественными числами, должен принимать в расчет эти ограничения.

Об этом непрестанно приходится заботиться в «численных расчетах», используемых не только в научных или инженерных проектах, но и, например, в финансовом моделировании, где также применяются вещественные числа и численные алгоритмы. Ошибка в каждой операции незначительна и может быть совсем не страшной для результата. Плохо, когда эта ошибка накапливается в процессе выполнения миллионов и миллиардов операций, что в конечном итоге может привести к серьезным ошибочным следствиям и искажению результатов.

Численное программирование требует особых подходов, позволяющих избегать подобных неприятностей. Рассмотрим простой пример – в одной из последующих глав мы встре-

тимся с методом интегрирования, приближенно вычисляющим значение определенного интеграла:

$$\int_{low}^{high} f(x)dx \approx \sum_{i=0}^{n-1} f(low + i*step)*step, \quad \text{где } n = \frac{high - low}{step}$$

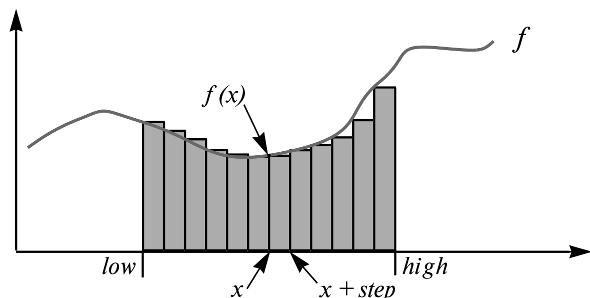


Рис. 10.3. Вычисление интеграла методом прямоугольников (конечная аппроксимация)

Алгоритм может быть реализован циклом. Покажем сейчас фрагмент алгоритма, а его полная версия появится позже. Во фрагменте используется локальная переменная *x* типа *REAL*:

```
from x := low until x >= high loop
    Result := Result + f.item ([x]) – f.item ([x])дает  значение f(x).
    x:=x+step                                                        [1]
end
```

В принципе, все работает, но обратите внимание на то, как *x* обновляется на каждой итерации. На каждом шаге добавляется переменная *step*, что приводит к появлению незначительной ошибки, но от итерации к итерации эта ошибка накапливается и может стать причиной существенного отклонения в процессе общего вычисления.

Некоторые программисты используют форму вычислений, подобную **[1],** полагая, что исключение умножения ускорит вычисление. Это, может быть, и справедливо, но эффект от накопления погрешности опасен. Разумная реализация вышеприведенной формулы использует умножение:

```
from x := low until x >= high loop
    Result := Result + f.item ([x]) – f.item ([x]) дает значение f(x).
    i := i + 1 ; x := low + (i * step) -                             [2]
end
```

Прямое вычисление *x* на каждом шаге позволяет избежать накопления погрешности. В худшем случае мы получаем ошибку, возникающую в результате выполнения одного сложения и одного умножения. Вот общий принцип:

Почувствуй методологию

Вычисления с вещественными числами

При разработке ПО, включающего действия с вещественными числами, учитывайте приближенный характер вычислений. При проектировании алгоритма избегайте накопления вычислительных погрешностей.

В краткой форме этот совет звучит так: изучайте вычислительную математику – часть прикладной математики, имеющей дело с численными вычислениями в противоположность символьным вычислениям. В этом курсе подробно рассматриваются свойства и ограничения представления чисел и операций, выполняемых на компьютере.

В ненадежной стране численного программирования существует надежное место: стандартизация. Ранее каждая компьютерная система имела собственную систему представления чисел и операций, из-за чего нельзя было гарантировать, что математически корректный алгоритм будет корректно работать на компьютере. Ситуация исправилась с введением стандарта IEEE на архитектуру арифметики с плавающей точкой. Этот стандарт определяет единые рамки для системы чисел компьютеров как с 32-битной, так и с 64-битной архитектурой. Большинство современных компьютеров соответствуют этому стандарту. Так что, если нужно проверить, что алгоритм не является причиной появления неприемлемых вычислительных погрешностей, то такую работу нужно проделать только однажды, полагаясь на стандарт.

10.2. О памяти

В памяти мы храним данные и извлекаем их по мере необходимости. На нижнем уровне представления в памяти хранятся символы, но для наших программ память представляется хранилищем, в которое можно помещать и получать объекты. Давайте разберемся, *что* память может делать в наших интересах.

Живучесть

На первом рисунке этой главы для изображения памяти используются два различных символа, отражающие два вида памяти с различными требованиями.

- Преходящая, кратковременная или оперативная, память хранит данные, создаваемые и обрабатываемые во время выполнения программы, но не гарантирует, что они сохранятся с окончанием этого сеанса выполнения. Технологически такая память при выключении электроэнергии теряет хранимые в ней данные.
- Постоянная память, обеспечивающая живучесть данных, сохраняет данные и при выключении энергии, при условии, конечно, что данные не удаляются явным образом.

Зачем нужна кратковременная память? Не было бы проще, чтобы все данные хранились вечно по умолчанию? Есть две причины, по которым это не делается, – технологическая и экономическая. Память, доступная процессорам во время выполнения, должна быть очень быстрой и, как следствие, является дорогой. Постоянная память значительно дешевле, так что ее можно использовать для хранения больших объемов информации – текстов, рисунков, музыки, расписания полетов, персональных данных и т. д., доступ к которым более медленный.

Слова: приемлемая или медленная скорость, большие или малые объемы, дорогая и дешевая – нельзя рассматривать вне контекста. Вот некоторые оценки (на момент написания оригинала книги).

- Компьютер, подходящий для разработки ПО, возможно, ноутбук, может иметь оперативную память в несколько гигабайт стоимостью в несколько сотен долларов за память. Время доступа к символу составляет примерно 50 наносекунд, что позволяет за секунду получить 20 миллионов символов (одна наносекунда – ns – равна 10^{-9} секунды).
- Компьютер может иметь **диск** (постоянную память) емкостью в сто раз больше оперативной памяти – в несколько сот гигабайт, стоящий вдвое дешевле, со скоростью доступа порядка 5 миллисекунд.

Как видите, время доступа к оперативной и постоянной памяти существенно различается, что непосредственно значимо для программиста. Программы, которые обрабатывают большие объемы данных, не могут игнорировать проблемы распределения и обмена данными между постоянной и оперативной памятью. Следует тщательно управлять временем передачи данных, чтобы сохранить приемлемое время выполнения приложения.

Оперативная память

Как отмечалось, операции процессора получают доступ к оперативной памяти. Этот ключевой компонент вычислений имеет несколько имен:

- *главная* память;
- *первичная* память;
- RAM-память (*Random Access Memory – память со случайным доступом*).

Последний термин имеет исторические корни. Вначале постоянная память, в отличие от первичной, была реализована технологически на магнитных лентах, а еще ранее – на перфорированных бумажных ленточках, где дырка в нужном месте задавала бит, равный 1, а отсутствие дырки означало, что значение бита равно 0. Доступ к данным на такой ленте был последовательным, так что время доступа к тому или иному биту зависело от его адреса; нужно было перемотать ленту, прежде чем прочесть значение бита. Оперативная память того времени была реализована как память прямого доступа. Время доступа к любому элементу не зависело от его адреса, поэтому такая память и получила название RAM-памяти. Теперь, конечно, и память на дисках является памятью прямого доступа, но название RAM все еще сохранилось для оперативной памяти;

- ферритовая (Core) память. Термин опять восходит к старым технологиям, когда элементами памяти служили ферритовые намагниченные сердечники – ферритовые ядра, так что до сих пор можно услышать, что множество данных хранится «в ядре».

На ниже представленной фотографии показана главная память – чип (микросхема), на два GB, изготовленная по технологии «DDR2_800», обеспечивающая время цикла в 5 наносекунд с пиковой производительностью передачи данных в 6,4 гигабайта в секунду.

Рис. 10.4. Микросхема с оперативной памятью

Разнообразие постоянной памяти

Существует два вида постоянной памяти.

- Некоторые элементы рассматриваются как часть компьютера и присоединены к нему постоянно. Они называются *вторичной* памятью, чтобы подчеркнуть тот факт, что они фактически являются продолжением первичной памяти. Их положительной стороной (помимо обеспечения живучести данных) является относительно низкая стоимость доступа, позволяющая сохранять сотни гигабайт. Обратная сторона медали – доступ значительно медленнее, чем к первичной памяти, и процессорам эта память непосредственно недоступна. Чтобы обработать данные, хранимые в этой памяти, они предварительно должны быть прочитаны в оперативную память.
- Другие элементы памяти присоединяются к компьютеру эпизодически на момент копирования данных, а затем могут быть удалены. Позже они могут быть присоединены к этому же или к другому компьютеру. В частности, они служат для хранения «резервных копий» данных. Они также называются **съемной памятью** или сменными хранилищами (storage) – синонимами памяти.

Наиболее общей формой вторичной памяти является диск. Более корректный термин – дисковод или дисковое устройство, включающее несколько дисков на одном стержне, вращающихся в процессе работы со скоростью от 4000 до 12000 оборотов за секунду. Данные считываются или записываются специальными головками, которые могут перемещаться над рабочей поверхностью вращающихся дисков. Значение считываемых или записываемых битов зависит от намагничивания небольших областей диска. Если выключить энергию, то диск вращаться не будет и операции записи и чтения становятся невозможными, но намагничивание остается, что и гарантирует сохранность данных на диске.

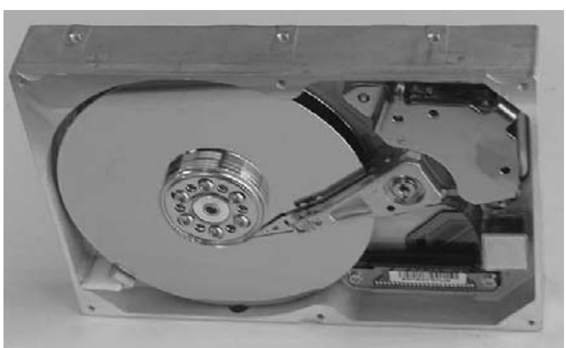


Рис. 10.5. Дисковод

Устройство, показанное на рисунке, имеет два диска, хотя вы видите только один. Оно может хранить 8 гигабайт (если это вас не впечатляет, то скажу, что это модель 1999 года, но я пока не намерен разобрать свой новейший дисковод, чтобы показать вам его фотографию. В магазине на момент написания этого текста нетрудно приобрести диск на несколько сот гигабайт стоимостью в 50$, терабайтные диски также вполне доступны). Дисковод на рисунке обеспечивал скорость вращения 5400 оборотов в секунду со временем доступа в 9 миллисекунд и максимальной скоростью передачи данных в 33 мегабайта за секунду. Время досту-

па и передачи указаны приблизительно, поскольку важной характеристикой является латентное время – время задержки, связанное с перемещением головки диска к нужной дорожке, после чего получить требуемые данные на дорожке можно много быстрее. Когда разрабатываются программы, в которых предусмотрена интенсивная работа с диском, нужно учитывать это свойство для оптимизации производительности.

Пока еще диски остаются доминирующим видом постоянной памяти, но уже появился серьезный конкурент в виде SSD-устройств, использующих флеш-память (SSD – Solid-state drive – полупроводниковый или твердотельный диск). В отличие от дисков SSD являются чисто электронными устройствами (а не электромеханическими, где перемещение головок обеспечивается механическим устройством) и не включают никаких подвижных частей. Они не подвержены «разрушениям диска», традиционным для «дисковой» технологии, когда в результате все данные на диске будут потеряны без всякого предупреждения.

Недостатком флеш-памяти является то, что она поддерживает только ограниченное число перезаписываний, хотя уже есть способы, позволяющие справиться с этим ограничением. Уже к концу 2008 года SSD стали привлекательной альтернативой дискам даже для ноутбуков, благодаря быстро возрастающей емкости и уменьшению цены.

Портативный компьютер – лэптоп, MIT Media Lab's XO laptop, – введенный в 2007 году как часть проекта «Каждому ребенку – свой лэптоп» (OLPC project – One Laptop Per Child), был одним из первых компьютеров, спроектированных для широкого распространения в мире, и использовал флеш-память, а не диски.

Рис. 10.6. Лэптор OLPC работающий с EiffelStudio

Следуя традициям бумажных лент, упомянутых ранее, некоторые устройства памяти являются съемными. Среди наиболее популярных являются USB-устройства, называемые так потому, что они связаны со стандартизованной последовательной шиной для передачи данных (Universal Serial Bus). Они используют технологию флеш-памяти емкостью от 2-х до 16 гигабайт, а теперь и больше. Съемными теперь являются и USB-диски, устроенные так же, как и встроенные диски, но присоединяемые к USB-порту, что и обеспечивает возможность их смены. Емкость таких дисков начинается от 100 GB.

Рис. 10.7. «Флэшка» и USB-диск

Регистры и иерархия памяти

Операциям, таким как сложение, требуются операнды, которые чаще всего должны находиться в специальных элементах памяти, называемых *регистрами*. Большинство архитектур имеют не более нескольких десятков регистров. Для выполнения операции над операндами, хранимыми в обычной оперативной памяти, например, для *a* и *b* в присваивании

```
a := a + b
```

необходимо выполнить следующие действия: прочесть значения *a* и *b* из оперативной памяти и записать их в соответствующие регистры, выполнить операцию над ними, в данном случае сложение, из регистра результата записать в оперативную память в область, отведенную для *a*.

Как результат, базисная иерархия памяти имеет три уровня:

- регистры, число которых мало, но скорость сравнима со скоростью центрального процессора;
- первичная память, ядро, емкостью в несколько гигабайт, но более медленная;
- диск или его эквивалент в несколько сот гигабайт емкостью, но еще более медленный.

Типичный порядок времени записи к моменту написания этого текста составлял: 0,5 наносекунды, 50 наносекунд и 5 миллисекунд. Цифры могут со временем изменяться довольно быстро, но порядок отношения обычно остается. Рассмотрим в частности отношение между двумя последними видами памяти, примерно равное 100000. Спроецируем эти отношения на человеческий уровень выполнения работ. Вообразите рабочего, который:

- способен обработать деталь, если она уже находится в нужном месте, за 0.01 секунды (аналог операций процессора, когда операнды помещены в регистры);
- получать заготовку из рядом расположенного хранилища, тратя на это одну секунду (аналог главной памяти, большой, но все же ограниченной);
- получать неограниченное количество заготовок от поставщика, удаленного за сотни километров, так что на доставку уходят целые сутки.

Для компьютера все цифры нужно делить на 20 000 000, но соотношение остается тем же самым. Политика управления памятью – что хранить в оперативной памяти, что на диске – существенно влияет на производительность.

Виртуальная память

На практике различие между главной памятью и дисковой затушевывается ввиду доступности *виртуальной памяти,* предоставляемой операционной системой. В результате можно

считать, что основной памяти значительно больше той реальной памяти, за которую мы заплатили.

Виртуальная память предоставляет программисту адресное пространство, значительно превосходящее физическое адресное пространство. Система управления памятью, стоящая за сценой, обеспечивает подкачку нужных данных с диска, когда они требуются, но не находятся в первичной памяти. Технически вся память разделяется на единицы, называемые *страницами*, каждая обычно размером в несколько килобайт. Фактический доступ к данным требует, чтобы они были в ядре, если это не так, то возникает *ошибка доступа к странице*, система виртуальной памяти в этом случае загружает соответствующую страницу из диска – эта операция называется «загрузка страницы» (page in). При загрузке страницы может возникнуть ситуация, когда физическая память использована и для страницы уже нет места. В этом случае предварительно произойдет «выгрузка страницы» на диск (page out). Есть разные сценарии, позволяющие выгружать страницы, которые с большой долей вероятности не понадобятся в ближайшее время.

Причина, по которой эта политика может быть эффективной, состоит в том, что в периоды интенсивной работы, обрабатывая циклы, программа использует небольшое подмножество – *рабочее подмножество* – из всего адресного пространства. Только в экстремальных ситуациях такая стратегия приводит к трешингу – режиму интенсивной подкачки, когда операции загрузки и выгрузки страниц беспрерывно сменяют друг друга.

Виртуальная память также облегчает режим разделения времени, когда компьютер выполняет одновременно несколько программ. Каждая программа в этом случае полагает, что у нее собственное адресное пространство, которое в реальности отображается на множество страниц, не пересекающихся в ядре, и на диске с адресными пространствами других программ.

Все это, конечно, затрудняет оценку производительности наших программ, так как реально не известно, в какой момент может возникнуть ситуация ошибки доступа к странице. Это похоже на ситуацию с нашим мифическим рабочим, который, обратившись в хранилище за заготовкой, вдруг обнаруживает, что оно пусто и требуются сутки, чтобы доставить деталь. Однако в большинстве практических случаев эти соображения можно игнорировать и полагать, что у вас реально есть, скажем, 20 GB, когда физически их только 4.

10.3. Команды компьютера

В обычных ситуациях командами, с которыми имеет дело программист, являются операторы языка программирования. Полезно, однако, понимать, в какие фактические команды компьютера транслируются операторы языка программирования – те команды, которые только и может выполнять компьютер.

Типичная команда компьютера хранится в слове памяти, иногда в нескольких словах. Она содержит код команды, определяющий ее тип – операцию, выполняемую командой, а также ноль или более требуемых аргументов, которые могут быть адресами или значениями. Для примера рассмотрим команду компьютера с архитектурой 32-битного Power PC. Команда занимает слово (32 бита) с нумерацией битов, начинающейся с нуля:

Позиция бита	0	6	11	16	21	22 … 31
Значение бита	0 1 1 1 1 1	0 0 1 0 1	0 0 0 1 1	0 0 1 0 0	0	0 1 0 0 0 0 1 0 1 0
Роль	Код команды	Результат	Операнд 1	Операнд 2		Вторичный код (266)

Код команды является комбинацией первичного кода 31 (двоичное 11111), заданного битами от 0 до 5, и вторичного кода 266 (двоичное 100001010) в битах от 22 до 31. Результат помещается в регистр 5 (двоичный код 101), а операнды читаются из регистров 3 и 4.

Побитовое представление неудобно для практического использования человеком. Обычная нотация заменяет двоичное представление более коротким – шестнадцатеричным (группируя биты в четверки, которые заменяются одной шестнадцатеричной цифрой) или восьмеричным (биты группируются тройками). Выше представленное слово 01111100101000110010000100001010 равно 7CA3210A в шестнадцатеричной системе, где буквы от A до F являются цифрами со значениями от 10 до 15.

Компьютеры обладают командами трех разных типов.

- Вычисления: арифметические операции, такие как сложение (в приведенном примере), вычитание, умножение, деление с вариантами для целых и вещественных чисел, операции сравнения, побитовые операции, которые выполняют логические операции над соответствующими парами битов двух слов. Обычно такие операции выполняются над операндами, расположенными в регистрах.
- Загрузки и хранения: передают данные из оперативной памяти в регистры и обратно, а также обеспечивают передачу данных с другими устройствами аппаратуры.
- Управления: переключая выполнение к другой точке программного кода в зависимости от выполнения некоторого условия или осуществляя безусловный переход.

Каждая команда имеет свой код, такой, как 31 в примере. Более удобно ссылаться на код команды, используя мнемонику. Для Power PC мнемоника команды сложения с кодом 31 – **add**. Представление программы в машинных командах не является удобной формой для общения с человеком. **Язык ассемблера** обеспечивает приемлемую, доступную для восприятия человеком форму представления таких программ. На языке ассемблера Power PC рассмотренная нами команда может быть записана так:

```
add r5, r3, r4
```

Язык ассемблера заимствовал у языка программирования возможность применения символических имен: имя регистра, такое как r5, имя команды, такое как **add**, – так же как использование идентификаторов для адресов и констант. Ответственность за трансляцию ассемблерного кода в машинный код ложится на программную систему, называемую *ассемблером*, которая представляет простейший вид компилятора. Трансляция облегчается тем, что для языка ассемблера выполняется соответствие «один в один» – одна команда ассемблера, как правило, транслируется в одну машинную команду компьютера. Операторы языка программирования задают куда более высокий уровень абстракции. Еще одна особенность ассемблера состоит в том, что для каждой архитектуры компьютера есть свой язык ассемблера, связанный с этой архитектурой. Современные языки программирования являются переносимыми (независимыми от платформы). Не будучи в полном смысле этого термина языком программирования, язык ассемблера позволяет преодолеть наиболее утомительные моменты, связанные с написанием и чтением программ в машинном коде.

Что должно остаться в памяти после всех этих обсуждений? Тот провал между командами, которые может выполнять компьютер, и теми задачами, которые мы хотим решать с помощью компьютера. Это доказывает важность языков программирования и объясняет сложность разработки ПО.

10.4. Закон Мура и эволюция компьютеров

При рассмотрении влияния производительности аппаратуры на программирование нельзя не учитывать ось времени. Чрезвычайные успехи информационных технологий идут рука об руку с прогрессом в разработке аппаратуры. В 1965 году появилась статья Гордона Е. Мура (Gordon Moore), сооснователя корпорации Intel, в которой он в чрезвычайно яркой форме описал этот феномен. Наибольшую известность получил так называемый «закон Мура», формулируемый следующим образом: «Число компонентов, размещаемых на интегральной схеме при сохранении постоянной стоимости, удваивается каждые 18 месяцев» (заметим, что сам Мур говорил о двух годах, но потом эта константа была уменьшена по результатам наблюдений до полутора лет). Есть несколько вариантов этого закона, но все они говорят об экспоненциальном росте. Эти утверждения не являются в полном смысле этого слова «законами», такими, как законы, открытые Максвеллом, Ньютоном или Эйнштейном. Это наблюдения о прогрессе индустрии в течение нескольких десятилетий – наблюдения, которые оказались пророческими и до сих пор продолжают быть применимыми. Удивительно, что нет никакой другой области человеческой деятельности, хоть сколько-либо напоминающей столь удивительную скорость роста. Автомобили сегодня не в тысячу раз быстрее, чем 20 лет назад.

В то время как закон, сформулированный Муром, говорил о плотности размещения элементов на интегральной схеме, а тем самым, и о скорости обработки, варианты этого закона говорили о том же феномене, справедливом и для многих других аспектов, – размере памяти, скорости доступа, стоимости различных устройств. Я помню, что был поражен (немногим более десятилетия назад), когда стоимость мегабайта дисковой памяти опустилась ниже одного доллара. Сегодня не многие готовы платить доллар за *гигабайт*.

Основной закон Мура не может быть беспредельно устойчивым, поскольку размещение все большего числа элементов в ограниченном пространстве приводит к росту излучаемого тепла, а кроме того, есть чисто физические пределы скорости распространения сигнала. В результате, как говорят некоторые компьютерные архитекторы: «число людей, объявляющих, что закон Мура перестал действовать, удваивается каждые полтора года». Фактически, закон продолжает действовать, но на новом уровне. Решение дают **параллельные вычисления**. Не нужно создавать процессор, работающий еще быстрее. Можно создать несколько процессоров, работающих параллельно. Многоядерная архитектура компьютеров становится общепризнанной. Проблема в том, что пока нет удовлетворительного решения, позволяющего программистам использовать все преимущества параллельной архитектуры. Но ни слова больше на эту тему. Эта проблема требует отдельной книги.

10.5. Дальнейшее чтение

Стандарт IEEE для арифметики с плавающей точкой, доступный по адресу: ieeexplore.ieee.org/xpl/freeabs_all.jsp?arnumber=4610935

> Специфицирует обмен, форматы и операции для двоичной арифметики и арифметики с плавающей точкой в программистском окружении компьютера.

John Markoff: *Faster Chips Are Leaving Programmers in Their Dust*, in *New York Times*, December 2007, доступна по адресу: tinyurl.com/5cstbq.

> Не являясь научной публикацией, эта статья дает ясное описание необходимости параллельной архитектуры для поддержания закона Мура. Описывает трудности парал-

лельного программирования. В течение многих лет Джон Марков был корреспондентом газеты «Нью-Йорк Таймс» в Силиконовой долине и играл важную роль в индустрии.

John L. Hennessy and David Patterson; *Computer Architecture*, Fourth Edition: A Quantitative Approach, Morgan Kauffmann, 2006.

Классический учебник по архитектуре компьютеров. Дополнения в последнем издании отражают последние тенденции, в частности, переход к параллельной архитектуре.

Рис. 10.8. Дэвид Паттерсон (2007)

10.6. Ключевые концепции, изученные в этой главе

- Внутреннее представление данных в компьютере использует двоичную систему.
- Базисной единицей данных является бит с двумя возможными значениями: 0 и 1. Биты группируются в байты, содержащие 8 битов, и слова, обычно из 8-ми или 4-х байтов (64-битная или 32-битная архитектура). Адреса измеряются в байтах. Целые и вещественные числа обычно хранятся в словах. Существуют также более компактные представления из одного или двух байтов. Символы представлены одним байтом (расширенный ASCII) или двумя байтами (Unicode).
- Измерение величин, отличных от единиц памяти, например скорости, всегда использует десятичные единицы, такие как кило (тысяча), мега (миллион), гига (миллиард), тера (10^{12}), пета (10^{15}). Для описания размеров памяти и адресов общей практикой, несмотря на официальные стандарты, принято использовать те же префиксы (кило и другие) для степеней двойки, начиная с $2^{10} = 1024$ и примерно равное $10^3 = 1000$.
- Представление целых в компьютере является точным, но только в конечном интервале.
- Представление вещественных чисел является приближенным. Арифметические операции являются обычно источником появления погрешностей. Реализация численных алгоритмов должна избегать накопления таких ошибок.
- Иерархия памяти включает регистры, оперативную память и устройства постоянной памяти, такие как диски и флеш-память. Операции процессора применимы к операндам, хранимым в регистрах. Доступ к регистрам – самый быстрый (менее наносекун-

ды), но число регистров невелико. Оперативная память на сегодняшний день имеет порядок нескольких гигабайт со временем доступа примерно в 100 раз более медленным, чем доступ к регистрам. Эта память при отключении от источника питания не сохраняет значения данных. Внешняя память – диски, флеш – в сто тысяч раз медленнее оперативной памяти, но существенно превосходит ее по объему, от сотен гигабайт до терабайтов, обеспечивая сохранность хранимых в ней данных.
- Машинный код представляет систему команд компьютера – операции нижнего уровня, непосредственно выполняемые компьютером, – вычисления над операндами в регистрах, обмен данными между разными уровнями памяти, передачи управления командам.
- Для компьютерной индустрии характерен экспоненциальный рост, известный как закон Мура. Поддержание этой тенденции потребовало перехода к многоядерным процессорам и параллельному программированию.

Новый словарь

Address	Адрес	**Bit**	Бит
Byte	Байт	**Core**	Ядро (Первичная память)
Disk	Диск	**Flash memory**	Флеш-память
Kilo	Кило	**Giga**	Гига
Hexadecimal	Шестнадцатеричный	**Mega**	Мега
Multicore (and manycore)	Многоядерный	**Moore's law**	Закон Мура
		Octal	Восьмеричный
Persistent	Живучий (сохраняемый)	**Primary memory**	Первичная (оперативная) память
RAM	RAM-память прямого доступа	**Read**	Чтение
Register	Регистр	**Removable memory**	Сменная память
Secondary memory	Вторичная память	**Storage**	Хранилище
Transient	Кратковременный	**Word**	Слово
Write	Запись		

10-У. Упражнения

10-У.1. Словарь

Дайте точные определения терминам словаря.

10-У.2. Карта концепций

Добавьте новые термины в карту концепций, построенную в предыдущих главах.

10-У.3 Измерения

Сколько байтов в:
- килобайте
- мегабайте
- мегаслове (слово = 4 байта)
- гигабайте?

10-У.4. Ваш новый лэптоп

Каталог рекламирует лэптоп с 1,3 GB памяти.

- Сколько байтов содержит эта память?
- Сколько битов содержит эта память?
- Предположим, что вся память используется для представления одной переменной. Сколько возможных значений может иметь эта переменная? Не требуется выписать точное число (подсказка: не пытайтесь это сделать, если только вы не являетесь владельцем бумажной фабрики; приведите аппроксимацию в форме 10^n).
- Если бы вы захотели написать это число на бумаге, 100 цифр в строке и 60 строк на странице, то сколько страниц вам бы потребовалось?

10-У.5. Размер и скорость передачи

Необходимо передать 128 MB данных, используя 128 Mb модем, работающий с максимальной скоростью. Сколько секунд это займет?

10-У.6. Восьмеричная арифметика

Восьмеричная арифметика использует систему с основанием 8 и цифрами от 0 до 7.

- Запишите десятичное число 300000 в восьмеричной системе.
- Число 74223 в восьмеричной системе запишите как десятичное число.
- Вычислите сумму этих двух чисел, используя восьмеричную арифметику. Представьте результат в обеих системах.

10-У.7. Шестнадцатеричная арифметика

Шестнадцатеричная арифметика использует систему с основанием 16 и цифрами от 0 до 9 и A до F.

- Запишите десятичное число 300000 в шестнадцатеричной системе.
- Число A42D3 в шестнадцатеричной системе запишите как десятичное число.
- Вычислите сумму этих двух чисел, используя шестнадцатеричную арифметику. Представьте результат в обеих системах.

11 Описание синтаксиса

При изучении структур управления и присваивания мы столкнулись с категориями языка, имеющими сложную синтаксическую структуру и допускающими вложенность одна в другую. Сейчас же встретимся с новыми интересными синтаксическими концепциями.

Для обоснования таких структур нам потребуется определение их синтаксиса. Первое представление структур управления использовало для их описания естественный язык, как например: «Условный оператор начинается ключевым словом **if**, за которым следует...». Такой стиль полезен для первого знакомства, но не позволяет задать общий способ спецификации — он многословен и недостаточно точен. Нам нужно нечто обратное – сжатость определения и математическая строгость.

Таким требованиям отвечает БНФ, Бэкуса-Наура форма (BNF – Backus-Naur Form), главный предмет изучения этой главы. Наряду с этой темой будут рассмотрены способы описания абстрактного синтаксиса, даны набросок разработки синтаксического анализатора (parser), введение в теорию конечных автоматов и кратко описана история вопроса.

11.1. Роль БНФ

Мы видели, что полное описание языка программирования включает три уровня: лексический, синтаксический, семантический. БНФ задают только спецификацию синтаксиса. Перед продолжением чтения следует освежить в памяти ранее введенные понятия – категория, терминал, нетерминал, образец, синтаксическое дерево.

Почувствуй историю

Первоначальная БНФ

История языков программирования началась в пятидесятые годы прошлого столетия. Первым языком, получившим широкое распространение, стал язык **Фортран (FORTRAN – FORmula TRANslator)**, предназначавшийся для научных вычислений и спроектированный командой из фирмы IBM под руководством Джона Бэкуса в 1954 году. В 1956 году для него был разработан компилятор, что предопределило успех и послужило толчком к созданию множества языков программирования.

Вскоре американские и европейские группы объединили усилия для проектирования международного стандарта языка, ставшего известным в 1956 году под именем Algol 58 (имя произведено от ALGOrithmic Language и вначале писалось буквами в верхнем регистре – ALGOL). Наибольшее распространение получила следующая версия этого языка – **Algol 60**.

При подготовке спецификаций обнаружилась необходимость лучшего способа описания синтаксиса, чем тот неформальный подход, который использовал Джон Бэкус при описании Фортрана. К этому времени Джон Бэкус входил в состав рабочей группы, создававшей язык Algol и предложившей нотацию, которая была известна первоначально как БНФ (Бэкуса Нормальная Форма). В 1964 году Дональд Кнут в письме в журнал «*Communications of the ACM*» просил учесть заслуги по разработке нотации другого члена комитета – Питера Наура из Дании, сохранив акроним БНФ, но изменив его расшифровку (Бэкуса-Наура Форма).
С тех пор было предложено много различных вариаций БНФ. В спецификациях языка Pascal – потомка языка Algol – его автор Никлас Вирт предложил графический вариант БНФ, который также стал широко использоваться.

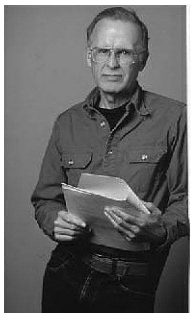


Рис. 11.1. Джон Бэкус

Рис. 11.2. Питер Наур (1995), портрет Дуо-Дуо Чанга

Языки и их грамматики

Для наших целей язык – это множество «предложений», каждое из которых задается конечной последовательностью **лексем** из некоторого «словаря». Например, простейшим правильным предложением языка Eiffel является текст класса:

```
class A end
```

Это предложение состоит из трех лексем: двух ключевых слов и идентификатора. Тексты, встречающиеся на практике, – тексты полезных классов – имеют значительно больше лексем.

Не каждая последовательность лексем из словаря языка является предложением этого языка: переставив лексемы **end** A **class**, мы не получим текст, задающий описание класса. Синтаксис языка и определяет, какие последовательности лексем являются предложениями языка, а какие нет. Спецификация синтаксиса называется **грамматикой**.

Грамматика

Грамматикой языка называется конечное множество правил, позволяющих создавать на основе словаря языка последовательности лексем, такие, что:

1) любая последовательность лексем, полученная в результате применения конечного числа правил, является предложением языка;
2) любое предложение языка может быть получено конечным применением правил.

Из определения следует, что любое предложение языка может быть выведено путем применения правил (утверждение 2) и что любой такой вывод дает предложение языка (утверждение 1).

Большинство языков потенциально бесконечно. Например, число возможных программ на Eiffel бесконечно. Эта теоретическая возможность не создает никаких практических проблем, во-первых, потому, что в нашей жизни мы можем иметь дело только с конечным множеством программ, но, что более важно, каждый текст класса – предложение языка Eiffel – является **конечной последовательностью** терминалов. Последовательность может быть очень длинной, но она не может быть бесконечной.

Конечное множество правил должно позволять порождать бесконечный язык, должно позволять, например, создавать описание всех возможных классов на языке Eiffel. Это опять-таки не должно нас беспокоить. Нам не нужны все возможные классы, нам нужны только те, что интересуют нас. Достаточно, что мы знаем, что правила способны породить описание каждого возможного класса.

БНФ – это нотация для определения грамматик. Это пример **метаязыка** – *языка*, служащего для описания других *языков*, таких как языки программирования.

БНФ и другие приемы, рассматриваемые в этой главе, применимы не только к языкам программирования, но и к любым формальным языкам, искусственным нотациям со строго определенной структурой. HTML, формат текстов Web-страниц, XML, формат структурированных текстов широкого применения, служат примерами формальных языков, не являющихся языками программирования в обычном смысле. Фактически, исходные исследования формализации грамматики языка были связаны с пониманием естественных языков с их сложностью и нерегулярностью.

Основы БНФ

Для описания грамматик будем использовать форму БНФ, называемую БНФ-Е, в частности, служащую стандартом описания Eiffel. Есть много других вариантов, таких как расширенная БНФ (EBNF- Extended BNF), определенная международным стандартом. В дальнейших обсуждениях термин БНФ будет применяться к любому варианту. Специфические свойства БНФ-Е будут отмечаться особо. Разница скорее в стиле, чем в существе дела.

БНФ позволяет нам задать грамматику языка – некоторую, а не вполне определенную грамматику, поскольку разные грамматики могут описывать один и тот же язык.

БНФ состоит из следующих частей, каждая из которых задается конечным множеством.

- Конечное множество **ограничителей,** к которым, как мы видели, относятся базисные ключевые слова (**class, if** ...) и специальные символы (точка, запятая ...).
- Конечное множество **категорий**, задающих структурные единицы языка. Примером категории является понятие Class, представляющее текст класса, и категория Conditional, представляющая текст условного оператора. Соглашение, принятое в БНФ-Е, требует, чтобы имя категории начиналось с большой буквы. Напоминаю, что конкретный пример категории называется **образцом**. Каждый конкретный условный оператор является образцом категории Conditional.
- Конечное множество **продукций**, где каждая продукция связана с некоторой категорией и задает форму ее образцов. Продукция для Conditional определяет форму любого условного оператора: вначале идет **if**, затем образец булевского выражения и так далее.

Каждая продукция определяет синтаксис образца категории через ограничители и другие категории. Вот пример продукции для категории Conditional:

```
Conditional ≜ if Then_part_list [Else_part] end
```

Это правило говорит, что любой образец Conditional – любой условный оператор – состоит из ключевого слова **if**, ограничителя, за которым следует образец категории Then_part_list, за которым, возможно, следует образец категории Else_part, ключевое слово **end** завершает конструкцию. Квадратные скобки отмечают возможные конструкции, которых может и не быть. Категории Then_part_list и Else_part имеют собственные продукции.

Каждая продукция определяет одну категорию, стоящую слева от символа ≜, который читается как «**по определению является**». В правой части этого определения стоит БНФ-выражение, задающее структуру образца категории. Такое использование продукций позволяет нам выделять два вида категорий.

- Категория, определяемая продукцией грамматики, называется **нетерминалом**.
- Другие категории являются **терминалами**. Примерами терминалов в грамматике Eiffel служат категория Idenifier (идентификатор) и Integer (целое), чьи образцы являются идентификаторами, такими как имя класса *Preview*, и целочисленные константы, такие как 34. Грамматика не определяет терминальные категории, их синтаксис определяется на более низком **лексическом** уровне.

Понятия терминала и нетерминала не являются новыми. Мы встречались с ними ранее при построении абстрактных синтаксических деревьев, где терминалы играли роль листьев дерева, а нетерминалы были внутренними узлами.

Причина отнесения некоторых категорий к терминалам и их определения вне грамматики – чисто прагматическая: эти категории имеют простую структуру, для которой мощь БНФ кажется избыточной. Идентификатор, например, – это начинающаяся с буквы простая последовательность букв, цифр и подчеркивания. Это правило может быть выражено лексическими приемами, изучаемыми в этой главе.

На синтаксическом уровне (БНФ-грамматики) образцы терминальных категорий задаются лексемами, подобно ограничителям. В отличие от ограничителей, каждый из которых представляет фиксированную лексему, такую как ключевое слово или специальный символ, большинство терминальных категорий, таких как Identifier и Integer, имеют бесконечное множество возможных образцов. БНФ-грамматика не интересуется содержимым лексем, рассматривая их как неделимые атомарные единицы.

Для любого языка особое значение имеет категория, описывающая структуру самого верхнего уровня; для Eiffel – это категория Class. Такой нетерминал (**top construct**) называется вершинной (начальной, главной) категорией языка. Предложения языка – тексты классов в нашем случае – являются образцами вершинной категории.

Отличия языка от метаязыка

Продукция, приведенная для категории Conditional, показывает, что БНФ включает символы трех разных видов.

- **Символы метаязыка**: они относятся к самой БНФ, позволяя описать структуру продукции. В пример с Conditional такими символами являются символ ≜ и квадратные скобки, сигнализирующие о возможности присутствия конструкции, заключенной в скобки.
- **Элементы языка**, непосредственно принадлежащие языку, который описывается грамматикой. К таким элементам относятся ключевые слова и ограничители, используемые в языке.
- **Имена категорий,** как терминальных, так и нетерминальных. Они принадлежат метаязыку, где обозначают элементы описываемого языка. Терминалы, как сказано, задают

лексемы. Нетерминалы задают синтаксические структуры. Например, каждый образец Conditional является синтаксической структурой, которая содержит подструктуру, такую как Then_part_list.

Во избежание недоразумений нужно быть внимательным и отличать символы языка от символов метаязыка.

В цветном оригинале книги используются разные цвета для символов разного вида. В черно-белом варианте контекст позволяет понять, какому языку принадлежит символ. Символов метаязыка относительно немного и они, как правило, отличаются по начертанию от символов языка. Для совпадающих символов – квадратных скобок, двоеточия – в БНФ символы языка Eiffel будут заключаться в кавычки.
Термин «образец» (spacimen) на первых порах мог вызывать недоумение. Казалось бы, следовало использовать привычный термин «экземпляр» (instance), но этот термин уже занят – он описывает объекты. Экземпляр класса – это объект, появляющийся во время выполнения, образец класса – это текст, задающий текст некоторого конкретного класса.

11.2. Продукции

Продукция определяет синтаксис одной категории. Она имеет следующую форму:

```
Construct ≜ Definition
```

В левой части продукции задается определяемая категория, а в правой – ее определение, выраженное в терминах категорий (терминалов и нетерминалов) и ограничителей. В зависимости от формы определения различают три вида продукций: конкатенацию, выбор и повторение.

Конкатенация

Конкатенация – это продукция, перечисляющая последовательность из нуля и более категорий; они следуют в определенном порядке и некоторые из них, возможно, заключены в квадратные скобки, что и определяет их возможность отсутствия. Наша первая продукция для Conditional является таким примером:

```
Conditional ≜ if Then_part_list [Else_part] end
```

Такая продукция задает, что каждый образец категории, стоящей слева, по определению состоит из последовательности (конкатенации) стоящих справа образцов, идущих в заданном порядке, с тем исключением, что «возможные» образцы могут отсутствовать.

Конкатенация просто означает связывание двух или более элементов в цепочку (catena в латыни). Это слово часто применяется в программировании. Мы говорим о конкатенации двух строк, когда вторая строка присоединяется к первой. Ее использование в БНФ немного претенциозно, поскольку можно было бы просто говорить о «последовательности». Но опять-таки в языке программирования мы уже задействовали этот термин, говоря о «последовательности операторов» – нашей первой структуре управления. Чтобы избежать недоразумений, мы используем термин «Последова-

тельность», говоря о конструкциях Eiffel, и термин «Конкатенация», когда речь идет о БНФ-продукциях. Аналогичное различие существует между терминами «Выбор» и «Условный оператор», «Повторение» и «Цикл». Первый термин используется при описании БНФ, второй – при описании конструкций языка программирования.

Выбор

Продукция «**Выбор**» перечисляет одну или несколько категорий, разделенных метасимволом вертикальной черты. Вот пример определения категории *Instruction* (оператор):

```
Instruction ≜ Conditional | Loop | Compound | Assignment | Call
```

Продукция «Выбор» задает, что каждый образец категории, стоящей слева, по определению состоит в точности из одного образца, стоящего справа. В отличие от конкатенации, порядок следования категорий, разделенных вертикальной чертой, не имеет значения. В последнем примере продукция говорит о том, что оператор языка может быть условным оператором или оператором цикла и так далее, перечисляя все возможные виды операторов языка. При чтении продукции вертикальная черта воспринимается и произносится как «или».

Повторение

Продукция «**Повторение**» перечисляет две категории, стоящие справа от символа определения: одну нетерминальную, которую следует повторить, другую – обычно терминальную, используемую как разделитель. Для примера определим категорию Compound (составной оператор), которая задает последовательность операторов, разделенных символом точка с запятой:

```
Compound ≜ {Instruction «;» …}*
```

Из этого определения следует, что образец составного оператора является последовательностью из нуля или более образцов операторов, каждый отделяется от следующего, если тот есть, символом «точка с запятой». В соответствии с этим правилом возможные образцы имеют вид:

- пусто (ни одного оператора);
- *inst1*
- *inst1; inst2*
- *inst1; inst2; inst3*
- «и так далее».

Здесь *inst1, inst2, inst3* являются образцами операторов.

Мы знаем, что в Eiffel точка с запятой, используемая в качестве разделителя, не является обязательной. Хотя это свойство может быть отражено грамматикой, более удобно правило грамматики оставить в приведенной здесь форме, дополнив его правилом, не использующим БНФ и устанавливающим безвредность пропуска разделителя.

В продукции использованы новые метасимволы, задающие повторение. Метасимвол * – символ, широко используемый в математике, означает «ноль или более раз». Метасимвол «многоточие» означает повторение элементов с разделителями, фигурные скобки используются для группировки элементов.

Такое продукционное правило для составного оператора с возможностью нулевого повторения означает, что допускается пустой составной оператор. Это может быть полезно в некоторых случаях, например, можно корректно писать:

```
if some_condition then                                                    [S1]
else
    instruction_1
    instruction_2
end
```

Здесь пустая **then**-часть законна, поскольку синтаксически здесь стоит составной оператор, пустой в этом конкретном случае. Этот пример можно переписать в более понятной форме:

```
if not some_condition then                                                [S2]
    instruction_1
    instruction_2
end
```

Но первая форма может быть предпочтительнее, учитывая процесс обновления программы, когда **then**–часть, предположительно, будет заполнена позже.

При определении некоторых категорий однократное присутствие образца обязательно, но возможно его повторение. В этом случае вместо метасимвола * (ноль или более) используется другой метасимвол, так же широко применяемый в математике для этих же целей, – символ $^+$ (один или более). Приведем теперь полное определение категории Conditional, включая определения входящих в нее категорий:

```
Conditional ≜ if Then_part_list [Else_part] end
Then_part_list ≜ {Then_part elseif …}+
Then_part ≜ Boolean_expression then Compound
Else_part ≜ else Compound
```

Продукция «Повторение», используемая в определении категории Then_part_list, показывает, что для образца категории допустимы такие формы:

```
cond1 then inst1
– Один образец Then_part
cond1 then inst1 elseif cond2 then inst2
– Два образца Then_part
cond1 then inst1 elseif cond2 then inst2 elseif cond3 then inst3
– Три образца Then_part
```

Пример можно продолжать, поскольку число образцов в этой конструкции может быть произвольно большим. Заметьте, что категория Then_part_list, задающая список, не может быть определена как возможно отсутствующая. В определении категории Conditional после **if** должна следовать хотя бы одна Then_part.

Правила грамматики

В БНФ – любом варианте – очевидное правило построения продукций состоит в том, что в правой части определения могут встречаться только ограничители, терминалы и нетерминалы. Для написания грамматики достаточно перечислить продукции, определяющие эти три множества, простыми соглашениями.

1. **Ограничители** описывают себя сами с дополнительным соглашением, что ключевые слова появляются в БНФ так, как они пишутся, а специальные символы появляются в кавычках.
2. Любой другой идентификатор, появляющийся в продукциях, обозначает категорию.
3. Если категория появляется в левой части хотя бы одной продукции, то он относится к **нетерминалам**.
4. В противном случае категория относится к **терминалам**. В этом случае ее определение должно появляться на лексическом уровне спецификации языка.

Случай 3 предполагает, что нетерминал может появляться в левой части более чем одной продукции. В БНФ-Е это не допускается, но разрешается для большинства вариантов БНФ, где две различные продукции, определяющие один нетерминал:

- A ≜ *Def1*
- A ≜ *Def2*

интерпретируются как одна продукция «Выбор»:

```
A ≜ Def1 | Def2
```

Варианты БНФ допускают в одной продукции смешение различных механизмов определения продукций – конкатенации, выбора, повторения, как в этом примере:

```
A ≜ B | C [D] {E «;» …}*
```

БНФ-Е отвергает такое смешение стилей.

Почувствуй методологию

БНФ-Е правило

- Каждый нетерминал должен появляться в левой части только одной продукции, называемой определением нетерминала.
- Каждая продукция должна быть одного вида: «Конкатенация», «Выбор» или «Повторение», следуя приведенным выше определениям.

Вышеприведенный пример определения категории А должен быть записан в БНФ-Е с использованием трех продукций:

```
A ≜ B | Concat
Concat ≜ C [D] Repet
Repet ≜ {E «;» …}*
```

Вместе с несколькими нотационными соглашениями это правило стиля задает специфику БНФ- Е, отличающую ее от других вариантов.

При написании определений языков я обнаружил, что это правило ведет к введению дополнительных понятий – нетерминалов, таких как Concat и Repet в последнем примере, а

ранее Then_part_list, что в конечном итоге позволяет выработать более понятное описание языка.

> Оно также позволяет дать лучшую оценку **размера языка**. Если допустимо смешивать различные виды продукций в одной, то можно иметь сравнительно небольшое число продукций, создавая видимость простоты языка. Так как этого нельзя делать в БНФ-Е, то число продукций является хорошим индикатором синтаксической сложности языка.

11.3. Использование БНФ

Все, что нужно знать о БНФ, уже сказано. Для эффективного использования этого метода описания языков рассмотрим некоторые прагматические наблюдения.

Применение БНФ

БНФ описание позволяет:

- понять синтаксис существующих языков (не только языков программирования);
- определить синтаксис языка, который предстоит спроектировать;
- написать синтаксический анализатор – парсер (*parser*).

Второе применение не столь фантастично, как кажется с первого взгляда. Возможно, вам еще не скоро придется проектировать язык общецелевого применения – соперник таких языков, как C#, Java или Eiffel. Но программистам довольно часто приходится иметь дело с «малыми» языками. Всякий раз, когда приходится обрабатывать данные сложного формата, эти данные можно рассматривать как предложения некоторого языка, синтаксис которого удобно задать, используя БНФ. Упражнения этой главы потребуют от вас выполнения подобной работы.

Третье приложение (построение анализатора) полезно при написании **компиляторов** и других инструментов, предназначенных для обработки программ, а в более общем случае – структурированных текстов. Одна из первых задач для таких инструментов – это реконструкция структуры текста в форме **абстрактного синтаксического дерева**. Это работа анализатора, как мы увидим в следующей главе. Любому анализатору необходимо формальное описание синтаксиса языка – он может получить его из БНФ-грамматики.

Язык, порождаемый грамматикой

БНФ-грамматику можно рассматривать двумя дополняющими способами, вытекающими из двух предложений в определении понятия грамматики.

- Это *механизм распознавания*, позволяющий определить, является ли некоторая последовательность из терминальных образцов и ограничителей фразой нашего языка, и, если это так, восстановить ее синтаксическую структуру. Эта точка зрения на грамматику полезна при написании анализатора.
- Грамматика является также *порождающим механизмом*: применяя ее правила, можно сгенерировать все фразы языка, одну за другой.

Вторая точка зрения на практике менее полезна, но в то же время важна. Давайте исследуем ее немного глубже. Для порождения всех возможных образцов любого нетерминала – в частности, начального (или, что то же, вершинного) символа грамматики – достаточно анализировать продукцию, определяющую этот символ (напомним, что в БНФ-Е такая продукция единственная).

P1 Для конкатенации – породить все возможные последовательности образцов перечисленных категорий, учитывая, что категории со статусом «возможные» могут как присутствовать, так и отсутствовать.
P2 Для повторения – породить все последовательности из нуля и более образцов (одну или более для знака +) указанной категории с заданным разделителем элементов последовательности.
P3 Если на любом из предыдущих шагов встретился нетерминал, применяйте тот же процесс для порождения его собственных образцов.
P4 Для выбора применяйте предыдущие шаги ко всем перечисленным категориям и собирайте все образцы, порожденные каждой из категорий.

Эти четыре механизма порождения предложений применяйте до тех пор, пока хоть одно из них будет применимо. В конечном счете будут порождены все предложения языка. Процесс этот обычно не завершается, поскольку, как мы видели, языки, представляющие интерес на практике, являются бесконечными.

Применяя этот процесс к нетерминалу A, чья продукция использует B, возможно, придется применять те же правила – на шагах P3 и P4 – к другим категориям.

Рекурсивные грамматики

Последнее наблюдение может вызывать некоторое опасение. Что, если, применяя процесс к *A*, мы должны будем применить его к *B*, а это приведет к тому, что мы снова встретим *A*?

Продукция для составного оператора является хорошим примером:

```
Compound ≜ {Instruction «;» …}*
```

Определение включает категорию *Instruction*, продукция для которой включает категорию *Compound* : ≜

```
Instruction ≜ Conditional | Compound | …Other choices …
```

Заметьте, что и определение категории Conditional включает категорию Compound. Если попытаться понять структуру Compound, разыскивая его образцы путем применения вышеприведенных правил, то кажется, что мы впадем в цикл – вывод, не имеющий смысла.

Определение понятия, в котором явно или неявно понятие определяется через само себя, называется **рекурсивным**. Рекурсия – использование рекурсивных определений – самая популярная вещь во всех областях программирования, и мы посвятим ей отдельную главу. Но уже здесь, где нет практически полезных грамматик, не использующих рекурсию, мы убедимся в важности рекурсивных грамматик.

Начнем изучение с небольшого примера. Рассмотрим мини-язык с тремя ключевыми словами **heads, tails, stop** (головы, хвосты и остановка). Других терминалов в этом языке не будет. Начальным нетерминалом будет категория Game, и вся грамматика задается тремя продукциями – выбором и двумя конкатенациями:

```
Game ≜ Head_start | Tail_start | stop
Head_start ≜ heads Game
Tail_start ≜ tails Game
```

Ситуация напоминает ситуацию с категориями Conditional, Instruction, Compound, определяемыми друг через друга.

Время теста!

> Можете ли вы породить образцы категории Game, принадлежащие языку, который порожден этой грамматикой? Что является образцами категорий Head_start и Tail_start? Постарайтесь дать ответ прежде, чем продолжите чтение.

Из-за рекурсии грамматика может показаться бессмысленной. Но будем прагматиками и спросим себя, нельзя ли использовать грамматику для **генерирования** образцов, применяя описанный выше процесс. Заметим, что в продукции для Game одна из ветвей, **stop**, является терминальной, что позволяет сгенерировать первое предложение (образец Game):

- **stop**

Но теперь мы можем использовать полученную информацию для генерирования образцов Head_start и Tail_start, определяемых через Game. Соответствующие продукции скажут нам, что **heads stop** является образцом Head_start, а **tails stop** – образцом Tail_start. Воспользуемся этими образцами в продукции для Game, получим два новых образца:

- **heads stop**
- **tails stop**

Получив эти образцы, и применяя тот же процесс, можно получить следующее множество образцов:

- **heads heads stop**
- **heads tails stop**
- **tails heads stop**
- **tails tails stop**

Этот процесс удвоения образцов можно продолжать. Становится понятным и общая конструкция образца Game: он представляет последовательность «голов» и «хвостов», идущих в произвольном порядке, она заканчивается терминалом **stop**. Нетрудно доказать, что любая такая последовательность является образцом. Немного сложнее доказательство того, что образцов другого вида для Game нет.

Очень простой язык этой грамматики с начальным нетерминалом Game можно рассматривать как все последовательности возможных исходов бросания монеты при игре в «орел или решка». Последовательность заканчивается в тот момент, когда игрок кричит **stop**. Этот язык можно описать и не рекурсивной грамматикой:

```
Game1 ≜ Throw_sequence stop           Игра ≜ Последовательность_бросков стоп
Throw_sequence ≜ {Throw …}+           Последовательность_бросков ≜ {Бросок…}+
Throw ≜ heads | tails                 Бросок ≜ Орел | Решка
```

Грамматика задана тремя продукциями, первая из которых является конкатенацией, вторая – повторением с пустым разделителем, третья – выбором.

> Этот пример показывает, что не имеет смысла говорить о том, что язык однозначно определяет грамматику. Один и тот же язык может быть описан разными грамматиками. Но с другой стороны, каждая грамматика вполне однозначно определяет язык, так что можно говорить о языке, порожденном грамматикой.

Применяя для генерирования языка продукционные правила, мы использовали стратегию, в которой терминалы предпочтительнее нетерминалов. Выбрав другую стратегию, можно впасть в бесконечный цикл, не сгенерировав ни единого предложения. Например,

начав с первой возможности для нетерминала Game, получим Head_start, а после его замены – **heads** Game. Многократно применяя эту же стратегию для Game, получим последовательность, в которой всегда присутствует нетерминал Game, что не дает создать предложение языка, которое по определению состоит только из терминальных символов.

Во избежание подобных ситуаций процессу генерирования языка необходимы подходящие стратегии, или *эвристики*, подобные той, которая применялась для выбора – если есть ветвь, содержащая только терминалы, то выбирается такая ветвь, в противном случае выбирается продукция, начинающаяся с лексемы.

Даже при наличии таких эвристик процесс генерации не создаст ни одного предложения языка, если его грамматика *полностью* рекурсивна. Для запуска процесса генерации необходимо, чтобы по меньшей мере некоторые выборы включали только лексемы. Грамматики, такие как

```
A ≜ A
```

или

```
A ≜ B
B ≜ A
```

бесполезны. Эти проблемы подробно будут обсуждаться в главе, посвященной рекурсии. Более тонким случаем являются грамматики, содержащие лексемы, но являющиеся леворекурсивными, как в следующем примере:

```
Instruction ≜ Compound | Assignment
Compound ≜  Instruction «;» Instruction
```

Для простоты в этой грамматике Assignment считается терминалом, определенным вне грамматики (по аналогии с лексемами, определяемыми на уровне лексического анализа). Эта грамматика имеет смысл и порождает такие предложения, как:

- *assignment_1*
- *assignment_1; assignment_2*

и так далее. Для получения конструктивного вида таких рекурсивных определений необходима общая теория, набросок которой будет дан позднее.

Одна из форм грамматики, позволяющая справиться с возникающими сложностями и написать простой синтаксический анализатор, называется **LL(1)**-грамматикой. Эта грамматика характеризуется тем свойством, что первого символа разбираемого предложения языка достаточно для однозначного выбора нетерминала, определяющего это предложение. Грамматика языка Eiffel близка к **LL(1)**-грамматике. Например, если первая лексема равна **if**, то соответствующий оператор может быть только условным (нетерминал Conditional); если первая лексема – **from**, то *оператор цикла* и так далее.

11.4. Описание абстрактного синтаксиса

Синтаксис, который мы изучали в этой главе, задавал **конкретный** синтаксис программы со всеми ее ключевыми словами, ограничителями и прочими деталями, которые играют важ-

ную синтаксическую роль, позволяя избежать двусмысленностей, но не несут никакой семантики. Ранее мы встречались с **абстрактным** синтаксисом, в котором эти детали отсутствовали, оставляя только те элементы, что несут за собой смысл.

Мы видели, как описать результирующую синтаксическую структуру, используя АСД (**Абстрактное Синтаксическое Дерево**), такое как ранее показанное дерево, задающее синтаксис нашего класса *Preview1*.

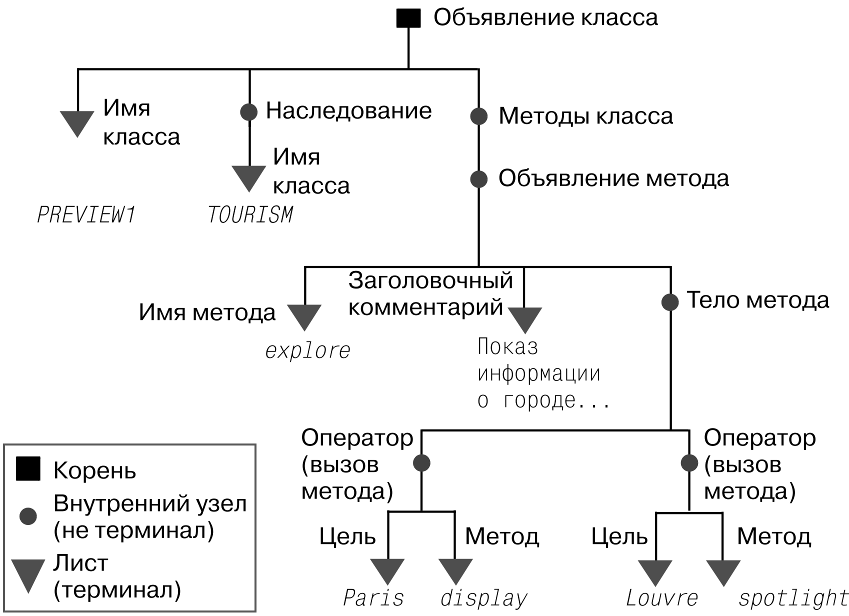


Рис. 11.2. Абстрактное синтаксическое дерево

Как отмечалось, проще построить конкретное синтаксическое дерево, содержащее все символы исходного текста. Некоторые компиляторы так и поступают, но обычно в этом нет необходимости. Для последующих фаз компиляции, таких как семантический анализ, генерация кода и его оптимизация, синтаксические маркеры не играют роли. Все, что нужно для представления структуры программы, в точности содержится в АСД.

Если бы нашей целью было описание абстрактного синтаксиса, без обращения к конкретному синтаксису, то нужды в новом формализме не было бы. Вполне достаточно использовать БНФ, опуская все лексемы, не являющиеся категориями лексической грамматики, в частности, опуская ключевые слова. Для последней продукции, специфицирующей Compound, точку с запятой можно было бы опустить, оставив просто Instruction Compound.

В таких приложениях, как синтаксический анализ и компиляция исходных текстов, эта грамматика не принесла бы пользы, поскольку, очевидно, здесь требуется конкретная грамматика, обсуждаемая до сих пор. Но она может играть свою роль, помогая в изучении тех структурных свойств текстов, которые не зависят от деталей внешнего вида этих текстов.

11.5. Превращение грамматики в анализатор

Одно из приложений БНФ, как отмечалось, в том, что грамматика является руководством для построения компилятора, начиная с фазы *синтаксического анализа*. Компиляторы, обычно являются **синтаксически управляемыми**: анализатор создает АСД, а последующие фазы компиляции продолжают работать на этой структуре данных, добавляя семантическую информацию (этот процесс называется *декорированием* дерева).

Детальное рассмотрение процесса построения синтаксически управляемого компилятора или просто анализатора выходит за пределы данной книги. При желании можно познакомиться с идеями применяемых методов, изучая библиотеку EiffelParse. В этой библиотеке реализован не самый эффективный механизм разбора, но ее методы представляют понятную и практическую иллюстрацию применения ОО-принципов этой книги для построения анализатора и компилятора. Сам Eiffel использует более традиционные подходы разбора, с которыми можно ознакомиться, изучая библиотеку «GOBO».

Идея, стоящая за EiffelParse, состоит в том, чтобы строить нужные классы непосредственно по грамматике БНФ-Е. Для каждой категории грамматики строится небольшой класс, являющийся наследником одного из классов библиотеки EiffelParse: *AGGREGATE, CHOICE, REPETITION* (соответственно для продукций «Конкатенация», «Выбор» и «Повторение»). Например, для конкатенации класс будет просто перечислять различные компоненты, стоящие в правой части продукции, связывая каждую компоненту с классом, подобным образом описывающим конструкцию. Следует быть внимательным, имея дело с левой рекурсией, но в остальном классы являются зеркальным отражением продукций БНФ-Е. Транслятор YOOC, разработанный Кристиной Мингинс, создает классы непосредственно по грамматике.

Для разбора входного текста достаточно вызвать EiffelParse – процедуру *parse* для соответствующей категории. В результате для нее будет создано АСД. Затем можно добавить *семантическую* обработку любого типа, используя методы синтаксического класса. Этот подход демонстрирует мощь и элегантность ОО-моделирования процесса анализа и компиляции языка программирования.

11.6. Лексический уровень и регулярные автоматы

Для терминальных конструкций, таких как идентификаторы и числа, БНФ не создает продукций, возлагая их спецификацию на лексический уровень. По этой причине терминальные категории называются также **лексическими категориями**. Их спецификация появляется в «лексической грамматике», дополняющей БНФ-грамматику.

Лексические категории в БНФ

На синтаксическом уровне, покрываемом БНФ, лексемы (терминалы и ограничители) являются атомами. На лексическом уровне нас интересует внутренняя структура этих атомов. Например (используя соглашения Eiffel):

- идентификатор – это последовательность символов, первый из которых является буквой (в верхнем или нижнем регистре), а остальные могут быть буквами, цифрами или знаком подчеркивания «_»;
- целое – это последовательность десятичных цифр (0-9), которые также могут содержать подчеркивание при разделении групп цифр в больших числах для облегчения чтения: 123_456_789;
- целочисленная_ константа (целое_со_знаком) – это целое, с возможно предшествующим знаком + или -.

Такие категории нетрудно выразить через БНФ (упражнение попросит вас проделать это). Но для таких простых конструкций обычно используют специфические лексические приемы, к изучению которых мы приступаем. Это позволяет избежать перегрузки грамматики продукциями для базисных структур, которые могут быть описаны более просто, и резервировать БНФ-грамматику для спецификации структур языка более высокого уровня, допускающих, в частности, вложенность.

Соответственно, компиляторы не используют анализатор (парсер) для разбора образцов терминалов, для этих целей применяется *лексический анализатор* (лексер).

Регулярные грамматики

Для определения структуры лексических категорий, таких как в вышеприведенных примерах, мы можем использовать **регулярную грамматику** – упрощенную версию БНФ.

Нетерминалами такой грамматики являются категории, подобные идентификаторам и целым, которые выступают в роли терминалов в БНФ. Для задания их структуры регулярная грамматика имеет собственные терминалы, обычно символы, принадлежащие некоторым категориям, например:

```
Letter ≜ ‘a’ | ‘b’ | ‘c’ | ‘d’ | ‘e’ | ‘f’ | ‘g’ | ‘h’ | ‘i ‘ | ‘j ‘ | ‘k ‘ | ‘l ‘ | ‘m’
‘n’ | ‘o’ | ‘p’ | ‘q’ | ‘r’ | ‘s’ | ‘t ‘ | ‘u’ | ‘v’ | ‘w’ | ‘x’ | ‘y’ |’z’
Decimal_digit ≜ ‘0 ‘ | ‘1’ | ‘2’ | ‘3’ | ‘4’ | ‘5’ | ‘6’ | ‘7’ | ‘8’ | ‘9’
Underscore ≜ ‘_’
```

Каждая категория выражается как выбор между **единичными символами**, показанными в одинарных кавычках. Такие категории являются по-настоящему терминальными (атомарными), не подлежащими дальнейшим уточнениям. Общепринято использовать специальную нотацию для последовательно идущих символов, учитывая порядок их следования в алфавите; так что продукцию для Letter, добавив еще буквы в верхнем регистре, можно записать в виде:

```
Letter ≜  ‘a’ ..’z’ | ‘A’ ..’Z’
```

Аналогично можно определить Decimal_digit как ‘0’.. ‘9’.

Регулярная грамматика может иметь те же виды продукций, что и БНФ, но со слегка отличными соглашениями и важными ограничениями:

- в продукциях «**Выбор**» можно использовать интервалы для задания множества символов;
- при определении лексической категории продукцией «**Конкатенация**» в последовательности символов не должно быть белых пробелов. Если вы определяете категорию как A B, то любой образец категории состоит из образца A, за которым следует без всяких разделителей образец B, никаких символов не должно быть между ними. Если языку требуется понятие разделителя, то его следует ввести явно в регулярную грамматику как лексическую категорию;
- **повторение** имеет упрощенную форму: если A означает ранее введенную категорию, то A^* и A^+ означают «ноль или более повторений A» и «один или более повторений A» соответственно. Опять-таки никаких разделителей или белых пробелов между образцами A не предполагается;

- **никакая рекурсия**, ни прямая, ни косвенная не допускается в грамматике. Простой способ выполнения этого запрета состоит в установлении *порядка применения* правил. Другой способ состоит в добавлении правила, согласно которому определение категории может ссылаться *только* на уже определенные категории.

В отличие от БНФ-Е регулярная грамматика позволяет смешивать различные виды продукций (поскольку на правила наложены существенные ограничения). Введение скобок позволяет устранить любую двусмысленность. Регулярная грамматика с учетом этих замечаний позволяет дать точные определения для рассмотренных нами лексических категорий:

```
Identifier ≜ Letter (Letter | Digit | Underscore)*
Integer_constant ≜ Decimal_digit+
```

Выражения, допускаемые только что определенными правилами, называются **регулярными выражениями**, а язык, определяемый регулярной грамматикой, – **регулярным языком**. Отметим следующее свойство.

Теорема: «Каноническая форма регулярного языка»

Любой регулярный язык может быть описан регулярной грамматикой, чьи продукционные правила не содержат в правой части никаких нетерминалов.

Доказательство следует из запрета рекурсивных определений. Как обсуждалось выше, любой появляющийся в правой части нетерминал определен в предыдущих правилах, поэтому вместо него можно подставить его определение. Понятно, что первое правило в такой грамматике содержит только терминалы в правой части, а в остальных правилах нетерминалы могут быть исключены, что и доказывает наше утверждение.

Например:

```
A ≜ T1 | T2 | T3*
B ≜ T4+ | A
C ≜ A B
```

Применяя процесс, описанный при доказательстве теоремы, можно построить эквивалентную грамматику, порождающую тот же язык.

```
A ≜ T1 | T2 | T3*      – Без изменений
B ≜ T4+| T1 | T2 | T3*    – Получено заменой A
C ≜ (T1 | T2 | T3*) (T4+| T1 | T2 | T3*)
```

Возможно, грамматика не стала более понятной, но свойство исключения нетерминалов в ней выполняется. Аналогично доказывается более сильное утверждение: любой регулярный язык может быть задан одним регулярным выражением. В нашем примере таковым является описание категории C, рассматриваемой как начальный символ грамматики.

Теорема высвечивает принципиальное ограничение регулярных языков: они не поддерживают рекурсивную вложенность. Мы видели, что язык программирования, по-

добный Eiffel, содержит условный оператор, где в качестве выполняемого оператора может быть любой оператор – условный оператор, оператор цикла и любой другой с неограниченной глубиной вложенности. В БНФ можно описать такие ситуации благодаря рекурсивно определяемым продукциям; с регулярными грамматиками этого сделать нельзя.

Зато регулярные грамматики удобны для задания правил описания лексем – первичных элементов языка. Когда нужно описать лексему, состоящую из одного или нескольких символов одного вида, за которыми следует один из трех специальных символов, за которым возможно следует последовательность символов еще одного вида, для таких ситуаций регулярная грамматика – то, что требуется.

Конечные автоматы

За кулисами регулярных выражений стоит математическая теория *конечных автоматов*. Для первого знакомства с этой теорией, о которой можно многое сказать, удобно воспользоваться визуальной иллюстрацией конечного автомата. Конечный автомат – это граф, с узлами, представляющими состояния автомата, и дугами, помеченными элементами некоторого базисного конечного множества. В нашем примере элементы представляют терминальные символы и имена категорий.

Следующий пример задает синтаксическую структуру квалифицированного вызова метода в Eiffel с возможными аргументами, подобно вызову *Line8.extend*(*new_station*):

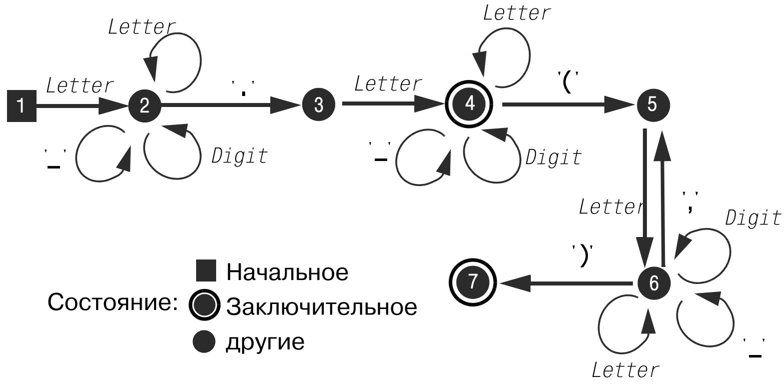


Рис. 11.3. Конечный автомат, распознающий вызов метода

Конечный автомат можно рассматривать как машину, обрабатывающую входную строку символ за символом. Процесс начинается в узле, задающем начальное состояние, затем продолжается, следуя выходящим из узла дугам, если таковые дуги имеются. Из выходящих дуг выбирается та, которая помечена очередным символом входной строки. Следуя дуге, попадаем в узел автомата, в который ведет выбранная дуга. Это называется **переходом** из одного состояния в другое. На входе *x9.f_g*(*a*,*a*), наш автомат стартует в состоянии **1**, входной символ *x* станет причиной перехода в состояние **2**, затем *9* станет причиной перехода в то же самое состояние **2.** Символ «точка» переведет автомат в состояние **3**, *f* переведет в **4**, подчеркивание и *g* оставят в **4.** Появление круглой открывающей скобки переведет автомат в со-

стояние **5**, из которого автомат, обрабатывая список аргументов, будет переходить в состояние **6** и снова возвращаться в **5**. Появление закрывающей скобки переведет автомат в заключительное состояние **7**, у которого нет выходящих дуг.

Язык, распознаваемый конечным автоматом, – это множество всех строк, на которых автомат, начиная работать в начальном состоянии, переходит в конечное состояние, полностью прочитав строку. Строки *Line8.extend* (*new_station*) и *x9.f_g*(*a*, *a*) принимаются нашим автоматом и принадлежат языку, им распознаваемому. Строки *не принадлежат* языку автомата, если:

- автомат достиг состояния, в котором нет дуги, соответствующей следующему входному символу. Для нашего автомата такой может быть строка *a.b.c* (допустимая в Eiffel, но не допускаемая рассматриваемым автоматом); обработав начальную часть строки *a.b*, автомат перейдет в состояние **4** и остановится, поскольку в этом состоянии нет дуги, помеченной точкой – очередным символом строки. Заметьте, что отказ будет верен и для заключительного состояния, если входная строка не обработана полностью, например, для строки *x.f*(*a*),*a*;
- обработаны все символы входной строки, но автомат не достиг конечного состояния. Примером является строка *a*, приводящая в состояние **3**, которое не является заключительным.

Основная теорема, связывающая регулярные языки и конечные автоматы, утверждает, что любой язык, заданный регулярной грамматикой, распознается конечным автоматом. Верно и обратное утверждение, что доказывает эквивалентность регулярных языков и языков, распознаваемых конечными автоматами. Не доказывая эту теорему, проиллюстрируем ее, построив для нашего автомата регулярную грамматику с языком, определяемым последней категорией:

```
Identifier ≜ Letter (Letter | Digit | Underscore)*
Another_argument ≜ "," Identifier
Argument_list ≜ "(" Identifier Another_argument* ")"
Feature_call ≜ Identifier "." Identifier [Argument_list]
```

Вызовы методов, распознаваемые этой грамматикой, являются подмножеством возможных в Eiffel вызовов, где выражения для аргументов допускают, подобно операторам, вложенность, как в вызове *x.f*(*y.h(z.i*)). Вышеприведенная лексическая грамматика и связанный с ней конечный автомат не распознают такие вызовы, поскольку аргумент для них может быть только идентификатором. Как только мы выходим за пределы лексем, так сразу требуется вся мощь БНФ. Заметьте, соглашение БНФ-Е для Повторения, включающее возможность появления разделителя, делает более удобным определение категории Argument_list, позволяя определить эту категорию одной продукцией, в правой части которой стоит {Identifier "," ...}$^+$.

Вышеприведенное описание автомата является *детерминированным*, и вот в каком смысле: начальное состояние автомата единственно, и в каждом узле не может быть несколько дуг с одним и тем же приписанным символом. Поэтому в процессе распознавания, проиллюстрированном выше, у автомата на каждом шаге не более одного перехода. Для недетерминированных автоматов это ограничение снимается. Можно, однако, доказать, что оба вида автоматов распознают один и тот же класс языков.

Конечные автоматы обеспечивают основу создания лексических анализаторов, часть компилятора, ответственную за распознавание лексем. Фактически, не представляет особого труда определить конечный автомат по регулярной грамматике, а затем по этому опреде-

лению построить непосредственно программу, распознающую лексемы. Такая схема используется в лексических анализаторах.

Контекстно-свободные свойства

Теория формальных языков выделяет несколько уровней по степени сложности их синтаксиса:

- **регулярные языки**, определяемые регулярной грамматикой;
- **контекстно-свободные языки**, задаваемые грамматикой, правила которой описываются продукциями с возможной рекурсией, подобно БНФ;
- **контекстно-зависимые языки**, для которых таких продукционных правил недостаточно.

В качестве примера, показывающего, что контекстно-свободная грамматика не может выразить всех свойств, требуемых в большинстве языков программирования, рассмотрим *правило задания типа*. В типизированных языках, таких как Eiffel, требуется, чтобы для каждого использования сущности *x*, в выражениях, таких как *some_function* (*x*), или в операторах, таких как *x.some_procedure*, в охватывающем модуле – методе или классе присутствовало объявление сущности в форме:

```
x: SOME_TYPE
```

Это объявление говорит, что *x* является локальной переменной метода или его аргументом или полем класса, а тип сущности, используемой, например, в качестве фактического аргумента при вызове функции, должен соответствовать типу *SOME_TYPE,* заданному при объявлении аргумента. В противном случае программа неверна, и компилятор должен отвергнуть ее. Но это нарушение другого рода в сравнении с ошибками синтаксиса, такими как:

```
if c then a + b end
```

Здесь нарушаются правила БНФ-грамматики, требующие, чтобы после **if** следовал оператор, а не выражение (как в примере).

Можно привести массу примеров, когда образцы, удовлетворяющие БНФ-грамматике, являются ошибочными в языке программирования: один из простейших – это оператор присваивания *a:* = *b*, где тип *b* не соответствует типу *a*.

Контекстно-свободные грамматики и БНФ не могут описать такие правила. Чтобы справиться с ними, грамматика должна ощущать контекст – быть контекстно-зависимой. Правило контекстно-свободной грамматики БНФ позволяет всегда заменять нетерминал A цепочкой μ, состоящей из терминалов и нетерминалов. Правило контекстно-зависимой грамматики позволяет заменять нетерминал A цепочкой μ только в определенном контексте, который можно задать в виде цепочки α, определяющей левый контекст, и цепочки β, задающей правый контекст. Правило такой грамматики говорит, что цепочку $\alpha A\beta$ с нетерминалом A можно заменить цепочкой $\alpha\mu\beta$.

На практике нет формализма для контекстно-зависимых грамматик, сравнимых по простоте и практичности с БНФ. Поэтому разработчики компиляторов предпочитают:

- использовать регулярные грамматики для описания лексических свойств языка и построения лексических анализаторов;
- использовать БНФ для управления свойствами, не зависящими от контекста, но позволяющими описать вложенную структуру программы, и на этой основе разрабатывать синтаксические анализаторы;

- выполнять все другие проверки, требующие анализа контекста, – контроль типов и прочее, используя дополнительные механизмы. Иногда применяются специальные формализмы, например, атрибутные грамматики, а иногда – приемы, учитывающие конкретную ситуацию.

Почувствуй историю

Классы языков и грамматик

Языки классифицируются как регулярные (Тип 3), контекстно-свободные (Тип 2), контекстно-зависимые с неукорачивающими правилами (Тип 1) и неограниченные (Тип 0, для распознавания которых необходима Машина Тьюринга, другими словами, вся мощь языка программирования). Эта классификация пришла из статей, опубликованных в 1956 и 1959 годах профессором MIT Ноами Чомски (Noam Chomsky, по-русски часто произносится как Хомский) и Марком Шютценбергером (Marco Shutzennberger) из университета Paris. Хомский известен также как политический деятель, его научные интересы связаны с изучением структур естественных языков. Его работы открыли новое направление в лингвистике и доказали продуктивность в изучении и понимании формальных языков и языков программирования.

Рис. 11.4. Ноам Хомский (2005)

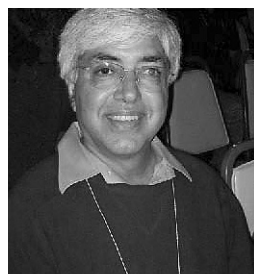


Рис. 11.5. Рави Сети (2008)

Рис. 11.6. Моника С. Лам (2008)

11.7. Дальнейшее чтение

1. Alfred V. Aho, Monica S. Lam, Ravi Sethi and Jeffrey D. Ullman: *Compilers: Principles, Techniques and Tools, Addison-Wesley, 2008.*

На русском языке: Альфред В. Ахо, Моника С. Лам, Рави Сети, Джеффри Д. Ульман Компиляторы: принципы, технологии и инструментарий, 2 издание, Издательский дом «Вильямс».

Последнее издание известного учебника по методам компиляции, остающегося стандартом в течение нескольких десятилетий.

2. Dick Grune, Henri E. Bal, Ceriel J.H. Jacobs and Koen G. Langendoen*: Modern Compiler Design; Wiley, 2000.*

Хорошее описание современной технологии построения компиляторов.

3. Steven S. Muchnick: *Advanced Compiler Design and Implementation*; Morgan Kaufmann, 1997. Еще одно описание важных методов построения компиляторов.

11.8. Ключевые концепции, изучаемые в этой главе

- Формальный язык, такой как язык программирования, представляет множество предложений, построенных на основе базисного словаря в соответствии с точными правилами.
- Большинство интересных формальных языков являются бесконечными.
- БНФ – это формализм описания формального языка конечным множеством правил, называемых продукциями.
- Каждая продукция грамматики БНФ описывает структуру некоторой категории или нетерминала – понятия грамматики, используя для описания другие категории и атомарные категории, называемые терминалами.
- Продукция определяет категорию либо как Конкатенацию других категорий, часть из которых может быть опущена, либо как Выбор между другими категориями, либо как Повторение другой категории.
- Компиляторы и другие инструменты анализа используют грамматику для декодирования – синтаксического разбора структуры входных текстов.
- БНФ позволяет также описать абстрактный синтаксис, который, в отличие от конкретного синтаксиса, не содержит ключевых слов и других элементов, не несущих семантическую нагрузку.
- Для элементарных конструкций входных текстов, таких как идентификаторы и константы, БНФ избыточна, их описание достигается более простыми средствами – регулярными грамматиками, чьи продукции не могут содержать рекурсии и не поддерживают вложенность. Регулярные выражения ассоциируются с конечными автоматами – математическими устройствами.
- БНФ покрывает класс контекстно-свободных языков, но не позволяет задавать свойства, зависящие от контекста, например, свойства контроля типа.

Новый словарь

BNF	БНФ	**Choice production**	Продукция «Выбор»
Concatenation production	Продукция «Конкатенация»	**Defining production**	Определение продукции
Grammar	Грамматика	**Lexical construct**	Лексическая категория
Lexical grammar	Лексическая грамматика	**Metalanguage**	Метаязык
		Phrase	Предложение
Production	Продукция	**Recursive grammar**	Рекурсивная грамматика
Repetition production	Продукция «Повторение»	**Top construct**	Вершинная категория (начальный символ грамматики)
Vocabulary	Словарь		

11-У. Упражнения

11-У.1. Словарь

Дайте точные определения терминам словаря.

11-У.2. Карта концепций

Добавьте новые термины в ранее построенную карту концепций для предыдущих глав.

11-У.3. БНФ для лексических грамматик

Напишите БНФ-грамматику, которая полностью описывает формы, применяемые в Eiffel для Identifier, Integer, Integer_constant.

11-У.4. Язык, определяемый рекурсивной грамматикой

Рассмотрите язык, определяемый рекурсивной грамматикой с вершинным символом Game.

1. Докажите, что любой образец в Game1 в нерекурсивной грамматике или, другими словами, любая последовательность из одного или более **heads** или **tails**, заканчивающаяся единственным символом **stop**, является образцом Game.
2. Верно ли, что любой образец Game является образцом Game1? Дайте ответ и обоснуйте его.

11-У.5. Регулярная грамматика с одной продукцией

Выпишите единственное регулярное выражение, которое полностью описывает язык, генерируемый категорией Game.

12
Языки программирования и инструментарий

За последние четыре десятилетия программный инструментарий коренным образом изменил способы проектирования и создания промышленных изделий – автомобилей и лекарств, газет, зданий, мостов, список можно продолжать. Этот инструментарий получил специальное имя – CAD (Computer Aids Design), CAM (Computer Aids Manufacturing), системы автоматизированного проектирования и автоматизированного производства.

ПО — это тоже промышленное изделие, и его производство требует проектирования. Опровергая старую поговорку «Сапожник ходит без сапог», программисты позаботились о создании инструментария, обеспечивающего их потребности, начиная от простых текстовых редакторов до интегрированных сред разработки, подобных EiffelStudio или VisualStudio от Microsoft.

Такой инструментарий является целостной частью профессиональной культуры, которой должен обладать разработчик ПО. Эта глава предоставляет обзор наиболее важных концепций. Мы начнем с концептуальных средств – языков программирования, и продолжим системным программным инструментарием, помогающим в построении пользовательского ПО.

Языки программирования предоставляют программисту способ выражения его алгоритмического мышления. У этих языков уже достаточно длинная история, существует не только много языков, но и много различных стилей. Мы коснемся истории развития стиля, принятого в этой книге, — ОО-стиля, а также рассмотрим базисные концепции стиля, характерного для функционального программирования.

Программный инструментарий предлагает широкий набор возможностей, начиная со средств, непосредственно связанных с языком программирования: компилятор и интерпретатор, которые можно рассматривать как близнецов-братьев. Мы поговорим о средствах подготовки программ – текстовых редакторах. Поговорим и о более мощном средстве автоматизированного проектирования ПО — инструментарии CASE (Computer Aided Software Engineering).

Отладчики, статические анализаторы и средства тестирования позволяют нам оценить и улучшить надежность программ. Средства управления конфигурацией позволяют нам сохранять историю развития компонентов ПО. Браузеры и средства документирования помогают понимать работу ПО на разных уровнях абстракции, что позволяет справиться с его сложностью. Метрический инструментарий дает нам количественные оценки качества разработки. Закончим это перечисление упоминанием IDE – интегрированной среды разработки, включающей, как правило, большинство из отмеченных средств и предоставляющей разработчику единые рамки для его многогранной работы. В последнем разделе главы будет дана краткая характеристика EiffelStudio, в частности, описан ее подход к компиляции – Технология Тающего Льда (Melting Ice Technology).

Обсуждение в этой главе носит общий характер, в нем меньше деталей, чем в предыдущих главах. Компиляция и интерпретация – это темы, заслуживающие отдельных курсов и учебников, здесь приводится только обзор. Рассматривайте эту главу, как неторопливую прогулку, некоторую разминку перед ожидающими нас подъемами, начиная с рекурсии в следующей главе.

12.1. Стили языков программирования

Языки программирования играют центральную роль в разработке ПО. Ядро этой книги использует язык Eiffel, который хорош тем, что позволяет в первую очередь сосредоточиться на концепциях программирования, а не на специфике конкретного языка. В приложениях приведена специфика четырех практически наиболее важных для практики языков: Java, C#, C++ и C. Сейчас же мы рассмотрим общие критерии, позволяющие классифицировать языки программирования, познакомимся с семейством ОО-языков и с другим популярным семейством, обладающим своими исключительными свойствами.

Критерии классификации

Языки программирования могут классифицироваться по разным критериям. Вот несколько наиболее важных.

- Область применения. Некоторые языки являются языками *общецелевого использования*. Другие ориентированы на определенную область, например, разработку Web-сайтов или разработку систем реального времени. Языки второй группы относят к языкам DSL (*domain-specific languages*), ПОЯ – *проблемно-ориентированным языкам*. Классификация языка со временем может быть подвергнута пересмотру, так как успешные языки в процессе развития преодолевают первоначальные замыслы. Примером является первый язык программирования Фортран, создававшийся как язык численного программирования (вычисления формул), а ставший общецелевым языком программирования в научных вычислениях. Другим примером служит язык Java, который первоначально создавался как язык для разработки ПО различных бытовых приборов, затем, с появлением Интернета, был переориентирован на создание апплетов — программ, загружаемых через Интернет, теперь же язык Java является ОО-языком общецелевого использования.
- Область действия программы. Некоторые языки предназначены для создания масштабируемых приложений (большой размер кода, много разработчиков, разработка и сопровождение ведется в течение многих лет). У других – цели более простые: небольшие разработки, проведение различных экспериментов, проверка гипотез. Так называемые Script (скриптовые) языки обычно относят ко второму типу. Конечно, нет гарантии, что первоначально небольшая программа в ходе эволюции не превратится в многофункциональную, долго живущую, большую программу. Как следствие, успешные языки второй категории в конечном итоге часто применяются к большим разработкам. Примером может служить офисное программирование на языке, встроенном в систему Microsoft Office.
- Возможность верификации. Некоторые языки проектируются так, чтобы компилятор мог найти потенциальные ошибки, — в результате анализа текста еще *до выполнения* можно будет гарантировать некоторые свойства периода выполнения. Обычно это накладывает на программиста дополнительные обязательства, так как верификация может требовать дополнительной информации (например, описания типов всех про-

граммных сущностей). Некоторые языки предпочитают простоту выражений, снимая требования и облегчая жизнь программиста в момент написания кода. Другое дело, что сделанные программистом ошибки проявятся в этом случае лишь на этапе выполнения.

- Уровень абстракции. Некоторые языки опираются непосредственно на ниже лежащий машинный уровень. Другие предпочитают использовать абстрактную модель вычислений.
- Роль жизненного цикла. Некоторые языки учитывают только проблемы реализации. Другие могут помочь на всех этапах жизненного цикла, на этапах моделирования системы, ее анализа и проектирования.
- Императивный или декларативный стиль описания программы. В императивных языках программы выполняют команды, изменяющие состояние программы. Дескриптивные языки по духу ближе к математике, позволяя описать требуемые свойства, но не выписывая точных шагов по их достижению.
- Архитектурный стиль. Он определяет то, как происходит декомпозиция системы на модули. Два главных подхода характерны для программирования – строить модули на основе функций или на основе типов объектов. Языки соответствующих двух стилей называются *процедурными* («функциональные языки» означает нечто другое, о чем позже пойдет речь) и *объектно-ориентированными* языками.

Для конкретного языка возможна почти любая комбинация этих характеристик. Стиль программ этой книги, иллюстрируемый программами на Eiffel, характерен и для таких языков, как C# и Java. Он предполагает:

- общецелевое использование (специализация допускается благодаря библиотекам), пригодность для больших разработок;
- возможность статического контроля текста на этапе компиляции, высокий уровень абстракции (с возможностью учета нижнего машинного уровня опять-таки с помощью библиотек);
- учет жизненного цикла (в частности, это верно для языка Eiffel, который повседневно используется для спецификации и проектирования);
- императивность и объектную ориентированность языка.

Другим примером является язык С, императивный, но не объектно-ориентированный, с довольно низким уровнем абстракции, но позволяющий управление выполнением на почти машинном уровне. Язык С++ добавляет объектный слой к языку С.

Продолжая обсуждение языков программирования, рассмотрим:

- введение в стиль языков программирования, относящихся к «дескриптивной» категории, – языков функционального программирования, радикально отличающихся от доминирующей практики сегодняшнего дня;
- некоторые основы ОО-языков.

Функциональное программирование и функциональные языки

В одной из предыдущих глав говорилось о принципиальном отличии математики от программирования (стоит перечитать это обсуждение).

- Программы работают на состояниях (состояние можно рассматривать как абстракцию понятия памяти). Они преобразуют состояние, выполняя, например, такой оператор,

как присваивание, изменяющий значение переменной в состоянии. В общем случае команды изменяют состояние. Изменение состояния называют часто *побочным эффектом*.

- Математический подход — чисто дескриптивный: здесь ничего не меняется, математика говорит о значениях и отношениях между ними.

Не все программисты готовы мириться с существованием такого отличия. Стиль, известный как функциональное программирование, старается приблизить программирование к математическому выводу насколько это возможно, часто отказываясь от понятия «состояние». Базисная концепция основана на том, что функция в математическом смысле — это механизм, определяющий способ получения результата по известным аргументам без всякого побочного эффекта. Ожидаемое преимущество такого подхода в том, что программы могут быть проще, а математические приемы позволят делать более надежные и ясные выводы о свойствах таких программ.

Будем использовать следующую терминологию.

Определения: императивный, аппликативный

Стиль программирования, основанный на изменении состояния, называется *императивным*. Синоним для противоположного стиля – *аппликативный*.

Большинство языков программирования, включая Eiffel, поддерживают императивный стиль. Для языка Eiffel характерно строгое разграничение команд и запросов — запросы здесь относятся к аппликативному стилю, не допускающему побочных эффектов изменения состояния. Функциональные языки в основе своей аппликативны. В основе – поскольку многие из них поддерживают несколько императивных отклонений для выполнения операций, императивных по своей природе, таких как операции ввода-вывода, операции с базами данных.

Первым функциональным языком, включающим и некоторые императивные свойства, был язык Лисп (Lisp – List Processing), разработанный Джоном Маккарти и представленный в 1959 году.

Рис. 12.1. Джон Маккарти

Центральной концепцией языка является список, записываемый в круглых скобках:

```
(A B C)
```

Заметьте, в Лиспе термин «список» используется в особом смысле, который отличается от списковой структуры, изучаемой в предыдущих главах. Списки Лиспа фактически близки к бинарным деревьям – структуре данных, которую мы будем изучать в отдельной главе.

Списки являются структурой данных (фактически, единственной структурой данных в Лиспе, достаточно общей для покрытия широкого разнообразия приложений). Более того, они также задают структуру программы. Если *A* обозначает функцию, то в вышеприведенном примере список обозначает применение этой функции к списку аргументов, заданному оставшейся частью списка, — в более привычной нотации этот список записывается в виде *A(B, C)*. Мощь, простота и элегантность идей вдохновила многих людей и привела к тому, что большинство ранних работ по ИИ (искусственному интеллекту) основывались на Лиспе. Язык и до сей поры продолжает развиваться и активно использоваться, имея многочисленных потомков, в частности, язык Scheme, сохранивший основные концепции.

Даже не детализируя остальные концепции языка, заметим, что в самой базисной концепции заложена возможность использования функционалов – функций высшего порядка, то есть таких функций, аргументами которых являются функции. Рассмотрим список:

```
((F A B)(G C) D)
```

Это список из трех элементов, первый из которых сам является списком с тремя элементами, второй – список из двух элементов. Можно рассматривать F как функцию от двух аргументов, возвращающую в качестве результата функцию от двух аргументов. Обозначим через H функцию, задающую результат применения функции F к ее аргументам, а через E — результат применения функции G к аргументу C; тогда представленное выражение списка в целом (H E D) может означать применение функции H к ее аргументам E и D. Это общий и замечательно мощный механизм.

В последующих главах мы познакомимся с механизмом *агентов* Eiffel, позволяющим достичь того же эффекта в рамках ОО-подхода; в его основе — теория лямбда-исчисления. Эта же теория лежит и в основе Лиспа и других функциональных языков.

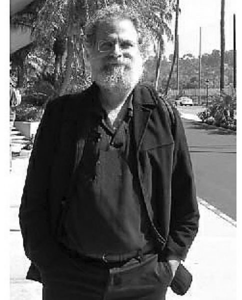


Рис. 12.2. Симон Джонс и Фил Вадлер

Для более близкого знакомства со стилем функционального программирования позвольте перейти от Лиспа к более современному и популярному языку Haskell. Основной вклад в разработку этого языка внесли Симон Пейтон Джонс и Фил Вадлер.

Ограничимся только одним примером, иллюстрирующим идеи функционального программирования, а также представляющим прекрасную подготовку для предстоящего изучения рекурсии.

Вспомним алгоритм обращения списковой структуры. Он включает манипуляции с указателем позиции списка. А нет ли способа получения результата, — ведь само понятие обращенного списка достаточно просто, — без обращения к таким деталям, как указатели позиции? Функциональное программирование такую возможность предоставляет, если вы согласны не обращать внимания на проблемы производительности и возлагаете их решение на «умный» компилятор. Список в Haskell записывается в квадратных скобках. Следующее определение задает функцию, обращающую список:

```
reverse:: [T] -> [T]
reversed [ ]                        = [ ]                                    [1]
reversed (first:remainder)    = reversed remainder ++ [first]               [2]
```

Это все, что нужно написать. Первая строка является описанием типа и говорит, что reverse – это функция, на вход которой подается список элементов типа T и которая возвращает в качестве результата список элементов того же типа. Определение этой функции дается в последующих двух строчках, представляющих разбор случаев.

- Первая строчка (случай [1] задает нерекурсивную ветвь определения), устанавливает, что обращение пустого списка является пустым списком.
- Вторая строчка (случай [2] задает рекурсивную часть определения) рассматривает обращение непустого списка.

Непустой список может быть всегда представлен в виде first : remainder, нотации, которая представляет список состоящим из непустого первого элемента – first, и остатка списка – remainder, который является списком, возможно, пустым. При таком представлении обращение списка может быть задано конкатенацией обращения остатка remainder и головы списка – first. Для обращения остатка рекурсивно может использоваться функция reversed. Заметьте, в Haskell для конкатенации применяется знак операции ++, при вызове функции не используются круглые скобки и вызов функции связывает сильнее (имеет больший приоритет, чем операция конкатенации). В более привычной математической нотации правая часть в **[2]** могла бы быть записана в виде: (reversed(remainder)) ++ [first].

В противоположность императивному стилю подобное определение функций является дескриптивным. Оно специфицирует свойства результата, не задавая точную последовательность шагов, которые нужно выполнить для получения результата. Это определение ближе к способу объяснения обращения списка, не предполагающего рассмотрения компьютерной реализации, — если список пуст, то он уже обращен, если же нет, то нужно обратить хвост списка и приписать в конце первый элемент.

Это определение аппликативно, здесь нет ссылок на состояния и их изменения, не вводятся переменные, которым программа может присваивать новые значения.

Хотя, как вы можете догадаться, в Haskell и других функциональных языках есть много других элементов, но и этот небольшой пример передает простоту и элегантность концепций функционального программирования.

Почему же мы все не переключились до сих пор на функциональное программирование? Вопрос для полемики – горячие сторонники функционального программирования считают, что мир должен переключиться, но следует отметить три важные проблемы.

- Производительность. Простота решения, продемонстрированного выше, может потребовать существенных накладных расходов — памяти и/или времени выполнения. Стоит заметить, что документация по функциональному программированию начинается обычно с приведения примеров, подобных рассмотренному, а затем следует рекомендация использовать более сложные варианты, обеспечивающие эффективность вычислений.
- Масштабируемость. Для структурирования больших систем понятие класса более эффективно, чем понятие функции. Следует заметить, что многие функциональные языки встраивают в язык ОО-конструкции.
- Отсутствие понятия состояния. Несмотря на то, что чисто аппликативные языки могут существенно упрощать понятие алгоритма, работая на возможно сложных структурах данных, некоторые аспекты вычислений фундаментально требуют введения понятия состояния. Уже упоминались операции по вводу и выводу данных, можно упомянуть и системы реального времени. Функциональные языки, Haskell, в частности, добавляют императивные аспекты программирования, но они не так просты, как базисная функциональная модель.

В глазах многих практикующих разработчиков ПО императивная объектная технология дает лучший ответ на критические вызовы, стоящие при разработке современных программных систем. Но в любом случае для всех разработчиков важно понимание концепций и приложений функционального программирования.

В императивных ОО-языках частично всегда возможно использовать функциональный стиль при описании рекурсивных функций, избегая при определении функций побочного эффекта. В качестве одного из упражнений в главе по рекурсии предстоит написать обращение списка в духе приведенного примера на Haskell.

ОО-языки

Прежде чем оставить тему языков программирования, позвольте остановиться на нескольких моментах, которые позволяют лучше понять стиль, лежащий в основе этой книги и доминирующий в индустрии ПО последние два десятилетия, — стиль ОО-программирования.

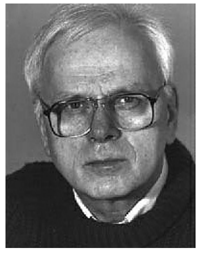


Рис. 12.3. Кристин Нигард и Улле Дал

Достаточно просто проследить за местом, временем и создателями этой технологии. Место — это город Осло в Норвегии, время – начало и середина 60-х годов прошлого столетия, создатели – Уле-Йохан Дал из университета Осло и Кристин Нигард из Норвежского вычислительного центра.

Совместно они спроектировали язык для *моделирования дискретных событий*. Первая версия языка – Симула 1 – предназначалась для моделирования на компьютере производственных процессов, рассматриваемых как последовательность событий. В 1967 году в журнале *Communications of the ACM* появилась их статья, посвященная описанию уже общецелевого языка программирования, где были введены основные идеи ОО-программирования. Прошло уже более сорока лет, но можно только удивляться, как много было предусмотрено в этой работе: классы, объекты, динамическое распределение памяти и сборка мусора, наследование (только одиночное), динамическое связывание и полиморфизм.

Странным образом эта работа настолько опередила время, что академическое сообщество оказалось незаинтересованным. Только несколько лет спустя появилась теория, поясняющая объектную технологию. Работа Парнаса по *скрытию информации* появилась в 1972 году. *Абстрактные типы данных (АТД)* были представлены в короткой статье в 1974 году Барбарой Лисков и Стефаном Зиллесом. Прочный математический фундамент АТД получили в диссертации Джона Гуттага в 1975 году.

Рис. 12.4. Барбара Лисков с Дональдом Кнутом

Первоначально, как было отмечено, язык Симула появился как язык моделирования. С этих пор одна из центральных идей ОО-подхода состоит в том, что программа является средством моделирования. Нигард в своих выступлениях часто использовал лозунг «Запрограммировать – значит понять» и гордился тем, что первый отчет о языке Симула назывался «Язык для Программирования и Моделирования». Моделирование дискретных событий с упором на описание внешних процессов представляло идеальную целевую область для разработки такой точки зрения. Эволюция от специализированной Симулы 1 до общецелевого языка Симула 67 показала общую применимость предлагаемых идей. ОО-концепции успешны, поскольку они эффективно поддерживают моделирование процессов в самых различных проблемных областях – банковских системах, обработке изображений, подготовке текстовых документов. Мы описываем такие системы, вводя соответствующие типы данных (*ACCOUNT, IMAGE, PARAGRAPH*), организуя иерархию наследования (*INTERNAL_ACCOUNT* как специальный случай *ACCOUNT*), применяя скрытие информации, что-

бы быть уверенными, что каждый такой тип можно разрабатывать независимо. Введение контрактов позволяет задать спецификацию таких типов объектов. Эти идеи, за исключением последней, были уже представлены в Симуле – ее создатели ясно осознавали потенциал языка.

В Европе появилось сообщество энтузиастов этого языка. Но не было мощной поддержки со стороны индустрии, многим этот подход казался экзотическим. В течение многих лет язык оставался наиболее охраняемым секретом индустрии ПО.

Но идеи со временем пробивают себе дорогу. В университете Юта – центре графических исследований – Алан Кей в своей диссертации объединил идеи Симулы и Лиспа. Он был приглашен в Исследовательский центр фирмы Xerox (PARC), расположенный в Пало-Альто, в Калифорнии – месте, где рождены многие замечательные новинки, аппаратные и программные, Там он создал первую версию языка Smalltalk и его программного окружения. Двумя другими ключевыми членами группы разработчиков были Адель Голдберг и Дан Инголс.

Рис. 12.5. Алан Кей, Адель Голдберг

Smalltalk – это динамически типизированный язык. В нем не найти объявления типизированных переменных, сплошь и рядом используемых в этой книге, а следовательно, нет защиты от несогласованности типов, которую компилятор организует для ОО-языков. Smalltalk внес значимый вклад не только в привнесение ОО-идей в язык, но и в разработку прекрасного графического интерфейса в среду разработки, в отличие от всего, что существовало в тот момент. Динамическая типизация, не давая гарантий надежности, взамен предоставляла разработчику высокую степень свободы, позволяя экспериментировать с самим окружением.

С появлением успешных версий Smalltalk 76 и Smalltalk 80 интерес к языку возрастал, но поклонники языка, подобно поклонникам Симулы, составляли скорее клуб, чем широкий поток в океане программирования. В 1981 году журнал *Byte* – флагман быстро растущего сообщества энтузиастов персональных компьютеров – решил опубликовать специальный номер, посвященный исключительно Smalltalk, хотя он не был доступен на компьютерах большинства читателей. В Августе 1981 года такой номер (теперь библиографическая редкость) вышел под редакцией Адель Голдберг, открыв широкую дорогу новому взлету ОО-программирования. Когда в 1986 году ACM организовал конференцию OOPSLA (Object-Oriented Programming, Systems, Languages, Applications, с тех пор ежегодную), ожидалось, что число ее участников будет представлять сотню человек, а оказалось, что оно перевалило за тысячу. Стали появляться новые языки. Бред Кокс использовал Smalltalk в ITT – мощной телекоммуникационной компании; Бьерн Страуструп из AT&T, рассматривавший Симулу в своей диссертации, решил расширить наиболее модный тогда язык С новыми идеями – так появились языки Objective-C и C++. Появление языка Eiffel также относится к этому периоду.

В течение нескольких лет эти языки преодолели первоначальное сопротивление индустрии, в частности, доказав эффективность реализации, и стали вначале важными игроками, а затем заняли доминирующие позиции. Позже, в 1995 году, появился язык Java, а еще через четыре года — и язык C#.

Со времен первой конференции OOPSLA критики предсказывают конец «эры объектов». Но никаких признаков такого исхода никогда не наблюдалось. Объекты продолжают процветать, как говорил один из героев – «в городе нет другой игры».

12.2. Компиляция против интерпретации

В оставшейся части главы анализируется инструментарий, поддерживающий разработку ПО. Программы не пишутся в машинном коде — в форме, доступной для непосредственного выполнения компьютером, они создаются на языке высокого уровня – языке программирования, ориентированном на человека. Конечно, язык программирования также можно рассматривать как машинный код некоторого абстрактного компьютера, отличающегося от реальных процессоров. Мы будем говорить в таких случаях об абстрактных или виртуальных машинах.

Целью компиляции является реализация возможности выполнения реальным компьютером кода, написанного для абстрактной машины. Компиляция, однако, — лишь одна из двух базисных технологий, применяемых для достижения нашей цели.

Базисные схемы

Вместо компиляции программы можно ее интерпретировать. Рисунок ниже иллюстрирует разницу подходов, игнорируя роль входных данных.

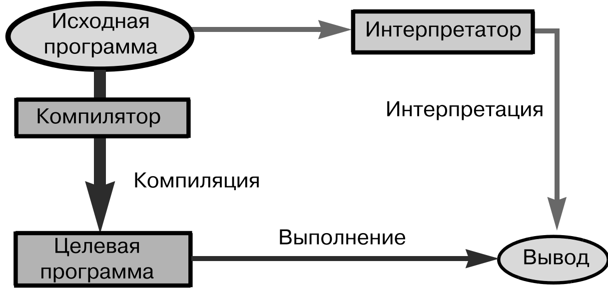


Рис. 12.6. Компиляция и интерпретация (без ввода данных)

И компилятор, и интерпретатор являются программами, на вход которых подаются программы, написанные на языке программирования. Компилятор транслирует переданную ему программу в целевую форму — машинный код, который уже может быть непосредственно выполнен на компьютере, создавая выходные результаты вычислений. Обработка той же исходной программы интерпретатором не создает целевой программы, но непосредственно выполняет ее, получая выходные результаты.

Интерпретатор должен быть способен определить эффект выполнения каждой конструкции языка программирования. Как пример того, как интерпретатор выполняет

свою задачу, рассмотрим интерпретацию присваивания $x: = x +1$. Интерпретатор должен хранить таблицу всех используемых переменных и связанных с ними значений. Он вычисляет новое значение x, добавляя 1 к старому значению, хранящемуся в таблице, а затем выполняет присваивание, заменяя старое значение x значением вычисленного выражения.

Компилятор будет генерировать машинный код, создающий тот же эффект, используя команды компьютера и адреса памяти (а не структуру более высокого уровня, такую как таблица).

> В дополнительном упражнении для небольшого языка потребуется, применяя эти идеи, написать компилятор и интерпретатор.

Если рассматривать язык программирования как машинный код для абстрактной машины, можно сказать, что интерпретатор – это программа, моделирующая вычисление на этой машине. Машинная память в этом случае также абстрактна и представляет структуры данных в виде таблиц интерпретатора «переменные – значения».

На следующем рисунке процессы компиляции и интерпретации дополнены вводом данных:

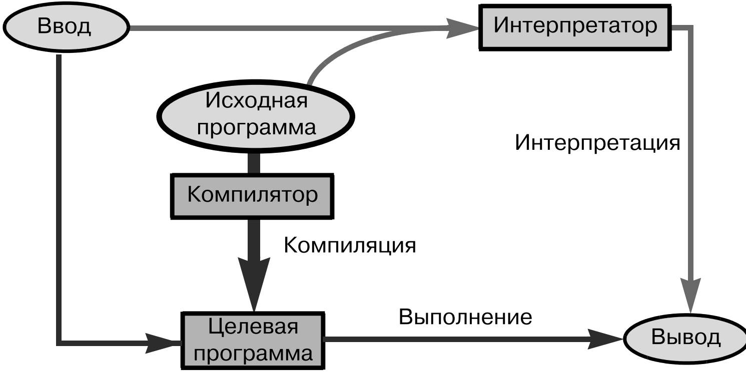


Рис. 12.7. Компиляция и интерпретация

Рисунок демонстрирует еще одну разницу между компилятором и интерпретатором. У интерпретатора два источника ввода – исходная программа и входные данные; а компилятору подается только программа. В последующем обсуждении этому различию придадим математическую форму.

Компилировать или интерпретировать? Эта проблема — предмет широко рассмотрения в компьютерных науках. Что лучше: непосредственно обрабатывать исходную информацию в том виде, как она есть, или предварительно привести ее к более удобной форме? Этот вопрос стоит не только при обработке программ, мы будем сталкиваться с ним и при изучении алгоритмов.

У компиляторов и интерпретаторов имеются свои достоинства. Возможны различные критерии. По производительности – времени выполнения программы – компиляторы побеждают.

- Выход компилятора является машинным кодом, непосредственно выполняемым компьютером. Дополнительно при создании этого кода компилятор мог применять оптимизацию, улучшающую эффективность кода.
- Интерпретация кода требует при выполнении каждого оператора его предварительной обработки. В результате интерпретация программы выполняется на порядок медленнее в сравнении с работой программы, созданной компилятором.

Все меняется, если в качестве критерия выбрать удобство и скорость разработки. Компилятор стоит между вами и реализацией вашей последней идеи: прежде чем увидеть результаты последнего изменения в программе, необходимо ждать результата компиляции (и связывания, о чем ниже пойдет речь). При интерпретации выполнение начинается незамедлительно.

> В современных средах разработки этот недостаток компиляции не является столь критическим благодаря применяемой технологии «*возрастающей компиляции*», когда при внесении изменений компилируются только те части программы, на которых это изменение сказывается. В конце этой главы мы рассмотрим, как эта технология работает в EiffelStudio.

Еще один критерий, согласно которому предпочтение опять-таки отдается компиляции, состоит в надежности программы. Компиляторы не просто транслируют программу — в процессе компиляции они осуществляют различные проверки, например, контроль типов для статически типизированных языков. Тем самым многие ошибки устраняются еще на этапе компиляции, в то время как для режима интерпретации ошибки обнаруживаются в процессе выполнения на одном из сеансов работы.

В принципе, интерпретатор также способен выполнять некоторые из проверок перед выполнением программы. Фактически, чистые интерпретаторы не применяются, они представляют собой всегда некоторую смесь компиляции и интерпретации.

Комбинирование компиляции и интерпретации

Схемы чистой компиляции и чистой интерпретации являются предельными вариантами: большинство практических решений является смесью. Это верно и для процесса компиляции в EiffelStudio, который будет рассмотрен позже в этой главе.

Заметим, что 100% схема интерпретации имеет мало смысла: каждый раз, когда интерпретатор выполнял очередной оператор, например, оператор цикла, он должен был бы возвращаться многократно к фактической последовательности символов и осуществлять ее разбор. Любое реалистическое решение не могло бы согласиться с такой неразумной тратой ресурсов. Так что фактически интерпретатор также начинает с преобразования входа в форму, приемлемую для интерпретации, например, строя абстрактное синтаксическое дерево. В ходе этого процесса, как отмечалось, возможен контроль проверки типов. Так что даже тогда, когда можно прочесть, что используется интерпретатор языка, частичная компиляция подразумевается.

Комбинирование интерпретации и компиляции идет значительно дальше рассмотренной основной идеи. Выход компилятора не должен быть непосредственным машинным кодом, он может быть субъектом дальнейшего процесса обработки.

Смешанная стратегия предполагает, что компилятор создает код на промежуточном языке, понимаемом некоторой виртуальной машиной – VM на рисунке. Такой подход объеди-

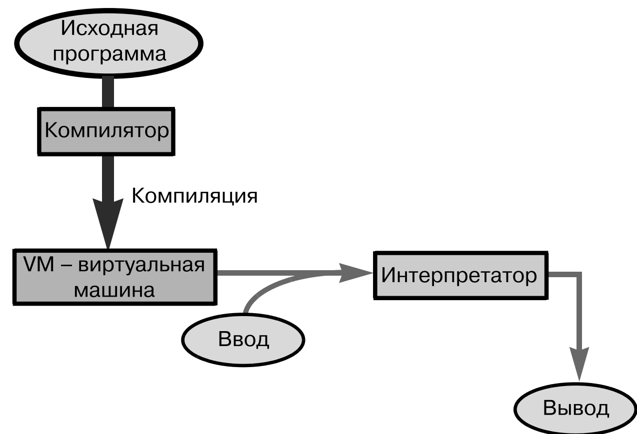


Рис. 12.8. Компиляция плюс интерпретация

няет преимущества компиляции и интерпретации. Благодаря тщательно спроектированной виртуальной машине возможно получить:

- переносимость, так как VM-код не зависит от специфики физических процессоров;
- повышение эффективности, поскольку создаваемый промежуточный код легко интерпретируется.

Виртуальные машины, байт-код и JIT (Just In Time) компиляторы

Реализация современных языков – Java, C#, других языков .Net — основана на смешанном решении. Промежуточный код для Java называется байт-кодом. В термине отражается тот факт, что виртуальная машина использует компактные команды, подобные командам фактического процессора, где каждая команда содержит код команды – типично задаваемый одним байтом, – после которого следует 0, 1 или 2 аргумента команды.

> Альтернативой байт-коду могла бы выступать виртуальная машина, непосредственно работающая со структурами данных, например, с абстрактным синтаксическим деревом для представления структуры программы и с хэш-таблицами для хранения свойств переменных. Но байт-код обеспечивает лучшую эффективность периода выполнения, как по времени, так и по памяти.

Прием двухэтапной компиляции был использован еще в семидесятые годы при реализации компилятора с языка Паскаль. Он получил второе рождение с распространением Интернета, так как хорошо был приспособлен для локального выполнения Web-клиентами. Поставщики апплетов – небольших программ – могли компилировать их в байт-код и поставлять их в такой форме. Дополнительным преимуществом к компактности стала переносимость кода, поскольку в противном случае машинный код пришлось бы создавать для каждой возможной целевой платформы.

Для выполнения апплета пользователям необходим только интерпретатор байт-кода. Они даже не должны знать, что такой интерпретатор существует, если он встроен в их Web-браузер, Поскольку при таком подходе возникают потенциальные риски, связанные с безопасностью, — жульнические или некорректные апплеты могут повредить ваш компьютер, — по этой причине для апплетов необходим интепретатор, который будет строго контролировать операции, разрешенные для апплетов.

Поставка программ через апплеты достигла некоторого успеха, но не стала основным способом распределенного ПО, как ожидалось в момент появления Java. Частично это связано с проблемами безопасности, но главная причина – в потере эффективности, возникающей по причине интерпретации. Большинство успешных апплетов являются небольшими программами, предназначенными для выполнения на Web-странице, включающие визуальную компоненту, при наличии которой потери времени представляются несущественными.

Для улучшения эффективности времени выполнения байт-кода применяются JIT (Just In Time) компиляторы, называемые джитерами, — осуществляющие компиляцию по требованию. Основная идея состоит в том, что машинный код для некоторого модуля создается «на лету», в тот момент, когда он первый раз вызывается на выполнение (не следует путать любителя джаза –jitterbug, с ошибками такого компилятора — jitter bug). Внесем соответствующие дополнения в предыдущий рисунок, который теперь выглядит так:

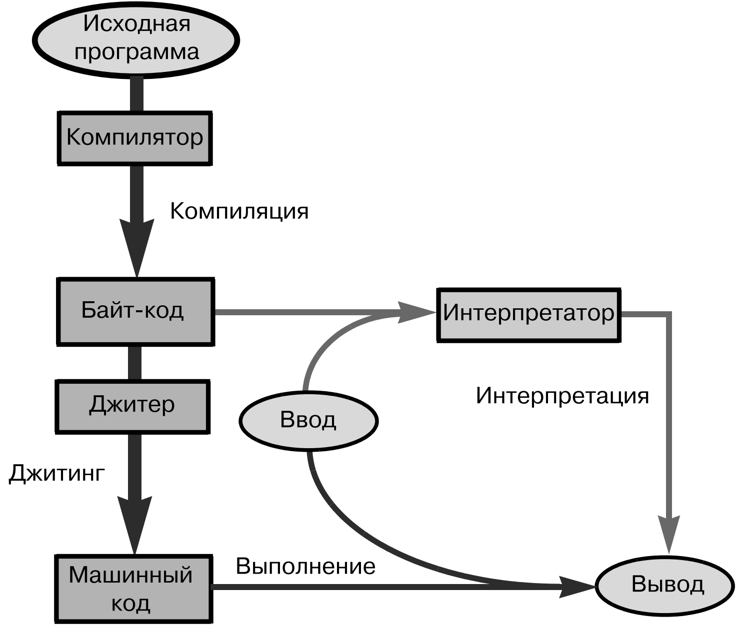


Рис. 12.9. Компиляция плюс интерпретация и джитинг

Обычно, как показано на рисунке, наряду с компиляцией «на лету» (джитингом) остается и возможность интерпретации байт-кода. Компиляция «на лету» обычно имеет место при первом использовании модуля (метода или всего класса), так что она будет нужна только для кода, фактически используемого в этом сеансе выполнения. В сравнении с традиционным компилятором, который компилирует всю программу, такой подход позволяет создавать более компактный код, сокращает время компиляции, но, что более важно, делает компиляцию частью процесса выполнения. Последнее является серьезным недостатком, поскольку к времени выполнения добавляются расходы на компиляцию, так что само время выполнения становится менее предсказуемым.

С первого взгляда кажется, что при таком подходе не стоит выполнять проверки типов и другой контроль, поскольку кому же хочется во время выполнения получать сообщения о нарушении согласованности типов? Это возвращало бы нас к проблемам динамически типизированных языков. Конечно, нам хотелось бы, чтобы все необходимые проверки выполнялись на первом шаге компиляции при создании байт-кода, так, чтобы любой код, передаваемый джитеру, был безопасным. К сожалению, эти утешительные предположения нереалистичны в распределенной среде, где опять возникают проблемы безопасности. Если вы загружаете байт-код из сайта, то можете ли вы знать, прошел ли он проверку? В общем случае — нет. Но тогда нарушения типа могут стать не только причиной нарушения надежности и аварийного завершения программы, все может быть гораздо хуже: в результате атаки становится возможным нарушение безопасности.

> С точки зрения специалистов по безопасности нарушения безопасности хуже аварийного завершения: при аварии все останавливается, при нарушениях безопасности программа может спокойно продолжить свою работу и даже дать правильные результаты, но при этом может быть утеряна конфиденциальная информация, стоящая дороже полученных результатов.

Как следствие, на практике компиляция на лету включает в любом случае проверку согласованности типов. Потери производительности при этом могут оставаться приемлемыми, поскольку система типов виртуальной машины с байт-кодом значительно проще, как правило, чем система типов исходной программы.

Стратегия компиляции в EiffelStudio также включает байт-код, но, как мы увидим, она использует различные способы комбинирования интерпретации и компиляции.

12.3. Основы компиляции

Сегодня компиляторы (и интерпретаторы) являются хорошо продуманными программными системами, которые вобрали опыт многочисленных исследований и разработок, продолжающихся уже 50 лет. Главная задача компилятора состоит в генерации кода для целевой машины, но это не единственная задача, как мы видели, — он должен проверять правильность программы.

Задачи компилятора

Компиляторы могут существенно различаться в деталях, но для всех вариантов есть общие задачи. Рассмотрим их примерно в том порядке, в котором компилятор должен применять их при обработке исходного текста.

Лексический анализ преобразует текст в последовательность лексем, представляющих идентификаторы, константы, ключевые слова и символы. Мы уже знакомы с базисными методами, используемыми при решении этой задачи: конечные автоматы и регулярные грамматики.

Синтаксический анализ воссоздает синтаксическую структуру программы.

Проверка правильности включает контроль типов и другие согласованные проверки. Eiffel, например, имеет примерно 90 «правил контроля правильности», таких как:

- в операторах присваивания и при передаче аргументов тип источника должен соответствовать типу цели;
- класс *B* не может назвать класс A своим родителем, если родителем *B* назван предок *A*. Это правило защищает от возникновения циклов при наследовании.

Семантический анализ включает обработку результатов, полученных на этапе синтаксического анализа, — структур данных, которые будут описаны ниже, таких как абстрактное синтаксическое дерево и таблица символов. На этом этапе создается важная семантическая информация, используемая на следующих шагах.

Генератор кода создает целевой код из исходного кода. Возможно, что на этом этапе будут выполняться несколько шагов по генерации кода, поскольку компиляторы могут использовать промежуточные представления, прежде чем сгенерировать окончательный код. Для исходного кода на Eiffel компилятор EiffelStudio генерирует байт-код, доступный для интерпретации (как часть технологии тающего льда, обсуждаемой ниже), но он также используется как промежуточный код, для которого компилятор может сгенерировать финальный целевой код.

Оптимизация улучшает процесс генерации кода, позволяя создавать более эффективный код. Оптимизация может встречаться в конъюнкции с некоторыми предшествующими задачами, такими как семантический анализ и генерация кода. Примеры оптимизации включают:

- распределение регистров – оптимизация периода выполнения. Математически $3 \times b + a$ имеет то же значение, что и $a + 3 \times b$, но одна из этих форм может выполняться быстрее другой из-за более эффективного распределения регистров. Оптимизация приводит к тому, что генератор кода выберет быстрейший вариант;
- удаление участков «мертвого кода». Если оптимизатор обнаружит, что некоторые участки кода программы никогда не будут выполняться во время исполнения, то он может удалить соответствующий сгенерированный код, а еще лучше — вообще не генерировать его с самого начала.

Программа, включающая никогда не выполняемые элементы, вовсе не обязательно означает программистские глупости. Если ПО, как и положено, основано на библиотеке повторно используемых компонентов, то простая стратегия компиляции может компилировать всю библиотеку, хотя сама программа на любом этапе ее разработки использует лишь часть этой библиотеки. В EiffelStudio, где большинство программ использует общецелевые библиотеки, такие как EiffelBase, удаление мертвого кода часто наполовину сокращает размер генерируемого кода.

Фундаментальные структуры данных

Задачи лексического и синтаксического анализа взаимосвязаны: «парсер» вызывает «лексер» (сокращения для синтаксического и лексического анализатора) для получения очередной лексемы. Главным результатом работы парсера является АСТ (абстрактное син-

таксическое дерево), которое задает структуру программы, очищенную от чисто текстуальных свойств, таких как ключевые слова. Другой фундаментальной структурой данных является таблица идентификаторов, в которой записаны имена, используемые в программе, — имена классов, методов, локальных переменных, других сущностей, — и свойства, связанные с каждым из этих имен. Например, для локальной переменной хранится тип этой переменной, метод, которому она принадлежит. Другие свойства, полезные для семантического анализа и оптимизации, могут включать списки операторов, использующих значение этой переменной, и списки операторов, модифицирующих ее значение. Хэш-таблицы, изучаемые в этой главе, хорошо подходят для реализации таблицы идентификаторов.

Для типичной организации современных компиляторов в задачу лексера и парсера входит создание АСТ и таблицы идентификаторов, инициализированной базисной чисто синтаксической информацией. Оставшиеся компоненты компилятора обогащают, часто говорят – декорируют, эти структуры данных более глубокой семантической информацией.

Проходы

Традиционное описание процесса компиляции включает понятие «прохода». На каждом проходе компилятор просматривает всю программу, выполняя специфические операции на ее компонентах. Важность этого понятия обусловлена еще и историей. На каждом проходе программа имеет свое представление – вначале это исходный текст, затем АСТ и так далее. В старые почтенные времена такие представления не помещались в оперативной памяти из-за ее ограниченных размеров и хранились во внешней памяти в виде отдельных файлов на диске, а еще раньше — на магнитных лентах. Компиляция состояла из последовательности проходов, каждый из которых обрабатывал ранее полученный файл и создавал новый. Естественное требование сокращения времени компиляции диктовало необходимость минимизации числа проходов.

Эти соображения влияли даже на проектирование языка. Паскаль, например, был явно спроектирован со строгим ограничением на «ссылки вперед»: не допускается в языке вызов процедуры, предшествовавший ее описанию (вначале опиши в тексте, а потом можешь вызывать). Такое ограничение позволяло выполнить однопроходную компиляцию.

Сегодня ситуация другая и понятие прохода менее четкое. В большинстве применяемых схем четко выделяется первый проход, на котором комбинация лексера и парсера создает АСТ и таблицу идентификаторов, а далее в процессе компиляции эти структуры данных обрабатываются и декорируются.

Компилятор как инструмент верификации

Всю значимость компилятора можно оценить, если вспомнить уже неоднократно высказанную мысль, что компилятор в дополнение к своей главной роли является еще и инструментом верификации. Для современных строго типизированных языков свойства типов предоставляют важную семантическую информацию. Правила проверки, используемые в Eiffel, описывают множество свойств, которые должны быть согласованными, что позволяет улучшить надежность создаваемого ПО. Выполняя проверку правильности, компилятор требует, чтобы эти правила выполнялись, обеспечивая тем самым обнаружение ошибок еще на этапе компиляции.

Исследования, выполненные в инженерии программ, показали все преимущества раннего обнаружения ошибок. Стоимость ошибок, обнаруженных динамически во время выпол-

нения, гораздо выше стоимости статически обнаруживаемых ошибок. С этой точки зрения ошибки времени компиляции – это хорошие новости.

Загрузка и связывание

Программам в машинном коде необходимы адреса памяти. Присваивание *x:* = *expr* будет помещать значение выражения *expr* по адресу памяти, связанному с *x*. В условном операторе **if** *c* **then** *a* ...управление будет передаваться в зависимости от вычисленного значения условия *c* по различным адресам, связанным с соответствующими участками кода. Вызов метода *r*(...) или *x.r*(...) приведет к передаче управления по адресу соответствующего кода, а затем вернет управление в точку, следующую за вызовом метода.

Точные адреса памяти компилятору недоступны. На современных процессорах большинство программ выполняются параллельно. Специальная программа операционной системы – **загрузчик** – ответственна за запуск других программ. Всякий раз, когда необходимо запустить на выполнение программу, загрузчик должен выделить программе соответствующую память и разместить там программу и ее данные. Такая схема позволяет компилятору не включать в генерируемый код фактические адреса памяти.

- Когда обрабатываются элементы программы, *принадлежащие некоторому модулю*, например, методы класса, компилятор управляет только относительными адресами в области памяти, отведенной модулю. Например, когда он обрабатывает неквалифицированный вызов *r*(...), появляющийся в методе того же класса *C*, что и *r*, компилятору известно смещение кода для *r* внутри области, отведенной для *C*. Компилятору неизвестны соответствующие абсолютные адреса, которые могут быть установлены только в момент загрузки и могут меняться от одного сеанса выполнения к другому.
- Для элементов другого модуля адреса будут смещаться относительно начального адреса этого модуля. Если бы вся программа компилировалась полностью, то это соответствовало бы уже рассмотренной нами ситуации, поскольку компилятор мог бы определить раскладки для всех модулей. Но часто желательно допускать **раздельную компиляцию**, когда модули компилируются независимо друг от друга и только потом объединяются в единую программу.

Первая задача имеет два возможных решения. Некоторые операционные системы применяют *распределяющий загрузчик*, который перед выполнением добавляет к каждому относительному адресу стартовый адрес модуля. Другое более общее решение: сама аппаратура спроектирована так, чтобы непосредственно использовать относительные адреса, интерпретируя адреса всех операторов относительно начального адреса.

Вторая задача требует применения еще одного инструмента операционной системы, называемого *компоновщиком*, или *линкером*, в задачу которого входит связывание отдельных независимо скомпилированных модулей в единый модуль. Любой из модулей, подаваемых на вход линкеру, может иметь так называемые *неразрешенные ссылки*, которые компоновщик будет заменять всюду, где это возможно, адресами, полученными из других модулей. Этот процесс может быть итеративным: если не все ссылки имеют целью уже связанные модули, то выход линкера может содержать еще не разрешенные ссылки, которые будут заполняться позже, на следующем шаге связывания.

Связывание является простой по концепции операцией, но может приводить к временным затратам, поскольку необходимо просматривать все множество модулей, подлежащих связыванию, что делает время связывания потенциально пропорциональным размеру программы. Желание справиться с этой проблемой привело к технологии «возрастающей ком-

пиляции», когда перекомпиляции подлежит только та часть программы, которую затрагивают сделанные изменения.

Время выполнения

Сегодняшние амбициозные языки программирования требуют не только изощренного инструментария в период компиляции (включая изощренный компилятор), но также серьезной поддержки на этапе выполнения. Когда программа выполняется, ей необходимо динамическое распределение памяти (для таких операторов, как конструкторы класса **create** *x*), нужна автоматическая сборка мусора, которая освобождает память от объектов, ставших недоступными программе, требуется обработка исключений и поддержка ввода и вывода. Аппаратура обычно напрямую не поддерживает эти механизмы. Эффективное управление памятью, в частности, основано на сложных алгоритмах и структурах данных.

Это потребности всех программ, а потому было бы неразумно, если бы компилятор генерировал соответствующий код для каждой программы. Вместо этого, когда программе понадобится одно из таких свойств, компилятор включает в генерируемый код вызов соответствующего метода из библиотеки, известной как система времени исполнения, или библиотека времени исполнения, или просто *исполняемая среда* (*the run time*). Программный код перед выполнением должен быть скомпонован с библиотекой исполняемой среды.

Другой способ установки роли исполняемой среды состоит в том, чтобы связать ее с понятием виртуальной машины. В то время как типичные машинные команды и видимые свойства «железа» — не считая скорости и размеров – не слишком изменились за последние пятьдесят лет, языкам программирования требуются более продвинутые виртуальные машины. Мы уже видели, что вполне возможно построить такую машину с собственными командами, например, байт-код, и создать компилятор, преобразующий исходный текст в код виртуальной машины. Полученный таким образом код может интерпретироваться, и недостаток такого подхода состоит в возможной потере эффективности. Другой возможный подход состоит в генерации кода для фактической аппаратуры, с обеспечением при этом более продвинутых механизмов исполняемой среды. В этом случае виртуальная машина представляет комбинацию аппаратуры и исполняемой среды.

> Виртуальные машины, основанные на байт-коде, также включают множество механизмов, поддерживающих выполнение байт-кода, так что понятие исполняемой среды применимо и к ним.

Для современных ОО-языков исполняемая среда необходима в той же степени, что и компилятор.

Отладчики и инструментарий выполнения

После того, как программа скомпилирована и скомпонована, возникает естественное желание запустить ее на выполнение. Окончательная версия программы типично является исполняемым *exe*-файлом. Но до завершения разработки необходимо контролировать выполнение, чтобы иметь возможность, например, в случае возникновения ошибки, исследовать контекст выполнения – анализировать содержимое объектов в точке ее проявления. Для этого необходим **отладчик** (*debugger*). Роль отладчика не только в том, чтобы отладить программу, разыскивая в ней ошибки (*bug*). Современные отладчики представляют инструмен-

тарий, предназначенный, как теперь модно говорить, для мониторинга за процессом выполнения программы.

Типичный отладчик включает такие средства, как задание точек останова (прерывания) в исходном тексте, запуск, прерывание, продолжение и завершение выполнения. Когда выполнение останавливается, что может быть вызвано одной из трех причин: достижением точки останова, возникновением ошибки или прерыванием, инициированным пользователем, — отладчик позволит исследовать код, приведший к текущему состоянию, проанализировать структуру объектов, видя их содержимое, динамически вычислять выражения, выполнять различные другие проверки программы и ее данных. В некоторых случаях отладчики, такие как отладчик EiffelStudio, позволяют выполнить откат по программе, чтобы повторно в пошаговом режиме проследить и понять причину, приведшую к такому состоянию (появлению ошибки). Следующий рисунок показывает типичное состояние отладчика EiffelStudio.

Наличие хороших отладчиков не дает нам права на «плохое» программирование (не надейтесь, что любая ошибка будет обнаружена в процессе отладки). Контроль выполнения может помочь только в некоторых случаях из несметного числа возможных сеансов выполнения.

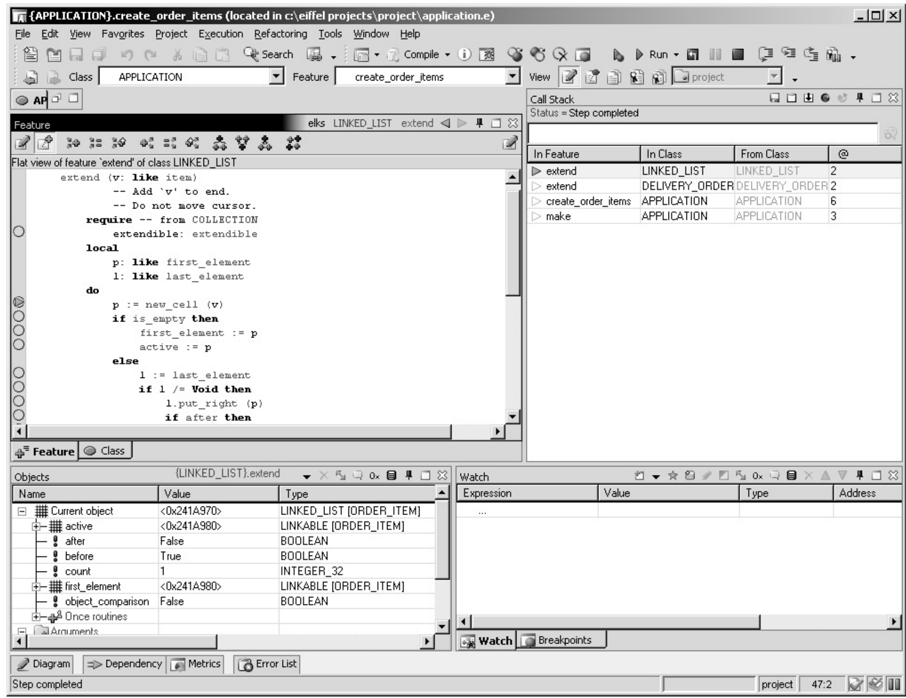


Рис. 12.10. Сеанс отладки в EiffelStudio

Методы динамической верификации, такие как отладка, не являются заменой статической верификации и проверки правильности. Всегда лучше избежать ошибки, чем исправлять уже допущенную. Отладчики важны, поскольку позволяют экспериментиро-

вать с программой и получать конкретное представление о ее поведении в момент выполнения.

12.4. Верификация и проверка правильности

Отладчики типично поддерживают проверку программы, выполняемую самими разработчиками. Перед тем, как работа над программой будет считаться законченной и она может быть передана пользователям, программа обычно подвергается систематическому процессу верификации и проверки правильности (verification and validation – V & V), который выполняется специальными людьми – тестировщиками.

Под верификацией понимают проверку внутренней согласованности, под проверкой правильности – проверку на соответствие внешним спецификациям. В главе, посвященной инженерии программ, процессу V & V будет уделено больше внимания. На данный момент просто стоит понимать, что соответствующие средства относятся к двум разным категориям.

- Статические анализаторы основываются на программных текстах. Примером является принуждение соблюдений правил согласованности типов и других ограничений правильности, выполняемых компилятором. Средства статического анализа на этом не останавливаются и идут далее в направлении возможного *доказательства корректности* программы.
- Динамические методы должны выполнять программу, тестируя ее на соответствие ожидаемым результатам.

12.5. Текстовые редакторы на этапах проектирования и программирования

Для ввода программных модулей, документов проектирования, других элементов необходимы текстовые редакторы – программы, позволяющие печатать и форматировать документы.

Когда документы являются программами, есть две возможности: использовать обычный текстовый редактор, пригодный для любого языка программирования, или специализированный редактор, знающий специфику языка, способный взаимодействовать с другими инструментальными средствами, такими как компилятор. Для каждого из подходов есть аргументы «за» и «против».

- Хороший общецелевой редактор предлагает много усовершенствованных свойств, например, способ выполнения сложных изменений или командный язык для обработки документов специальным образом. Но такие редакторы не «заточены» под программирование и не имеют некоторых характерных свойств, важных для программ. В частности, они не знают, что программы обрабатываются компилятором и выполняются под управлением отладчика.
- Специализированный редактор программ может получать преимущества, зная специфику языка, например, он может выполнять синтаксический анализ текста непосредственно в момент ввода этого текста. Он может иметь кнопку запуска компиляции и иметь другие способы прямого взаимодействия с различными средствами разработки. Но в отношении свойств, не связанных со спецификой языка, он может быть менее продвинутым, чем общецелевой редактор текста.

Технические рассмотрения не являются строго определяющими. Текстовые редакторы, такие как Vi или Emacs, завоевали много сторонников, и люди не хотят отказываться от привычных средств работы с текстами только потому, что данные тексты являются программами.

Еще одна причина, по которой редакторы программ не вытесняют общецелевые редакторы даже при вводе программ, состоит в том, что общецелевые редакторы могут быть параметризованными, что позволяет им поддерживать синтаксис специального языка программирования (или другую строго определенную нотацию, такую как HTML). Такому редактору можно передать определение конкретного языка в виде БНФ или эквивалентного формализма. В результате настройки редактор сможет обеспечивать преимущества, характерные для редактора программ: окрашивание текста для выделения ключевых слов и других элементов синтаксиса, автоматическое дополнение (если напечатать начале синтаксический образец, например, **if**, автоматически будет добавлен шаблон условного оператора с ключевыми словами, так что останется только ввести пропущенные элементы).

Эти наблюдения показывают, что среда разработки должна включать специализированный редактор, поддерживающий язык или языки в случае многоязыковой среды, но не ограничиваться этим — необходимо также принимать тексты, подготовленные другим инструментарием.

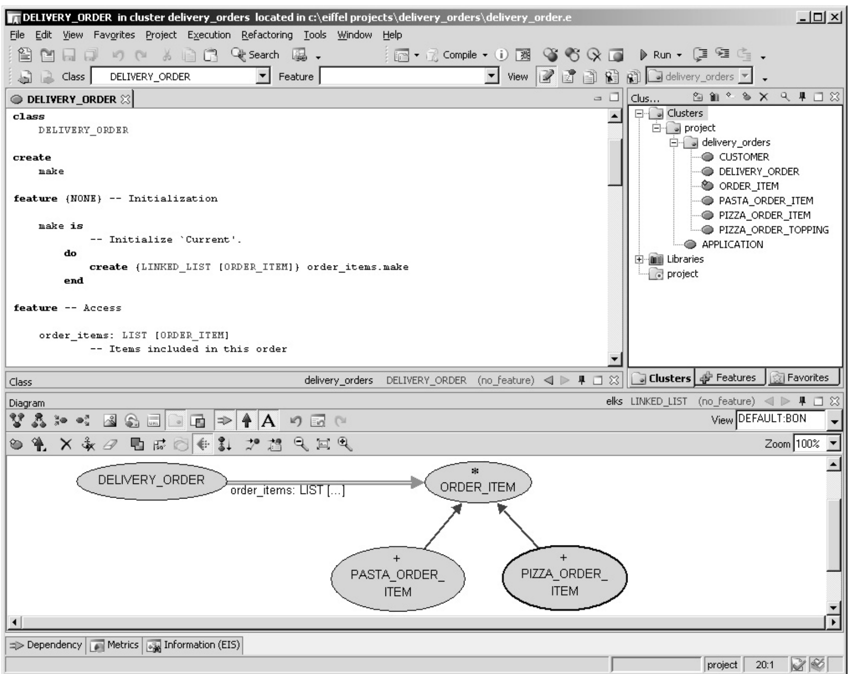


Рис. 12.11. Текстовый облик (вверху) и построение диаграмм в EiffelStudio

Как пример, EiffelStudio предлагает встроенный редактор для текстов классов. Работа стала бы проще, если бы можно было предположить, что это единственный способ для пользователей ввести текст класса, – было бы легче сохранять историю изменений, облегчался бы процесс перекомпиляции, описанный ниже. Но необходимо учитывать, что пользователи могут использовать другие текстовые редакторы. В этом случае компилятор должен анализировать файлы и их метки времени (время последней модификации), чтобы знать, что следует перекомпилировать.

Помимо чисто текстуальных средств часто может быть удобным применять графику для представления программных текстов, такую как диаграммы, используемые в этой книге для описания архитектуры ПО в виде множества классов со связями, которые отражают отношения наследования и «клиент-поставщик». Задействованная нотация называется BON-нотацией (Business Object Notation). Другой, более сложной нотацией, также широко используемой, является UML (Unified Modeling Language) – унифицированный язык моделирования. Графический инструментарий, поддерживая эти нотации, часто позволяет ввести диаграмму в интерактивном режиме, затем автоматически сгенерировать программный текст или его шаблон, отражающий структуру, которая задана диаграммой. Такие инструменты часто относят к CASE- средствам (Computer-Aided Software Engineering) — термин, который в буквальном смысле относится ко всем рассматриваемым в этой главе средствам, но обычно применяется в более ограниченном смысле. Хорошо известным инструментом, поддерживающим UML, является система Rose от Rational Software. EiffelStudio включает инструмент Diagram Tool, позволяющий отображать диаграммы классов и кластеров:

Такой инструментарий должен удовлетворять требованию «обращения цикла» — гарантировать, что преобразование графики в текст и обратно, выполняемое в любом порядке, возвращает оригинал. Для многих людей графические облики дают более точное понимание общей структуры. Но когда дело доходит до точной семантики, нет ничего лучшего, чем текст. Обращение цикла гарантирует согласованность этих двух точек зрения. Инструментарий позволяет вам свободно переходить от графического образа к тексту и обратно, изменяя по желанию либо картинку, либо текст, сохраняя согласованность представлений. Этот принцип соблюдается при работе с инструментом Diagram Tool.

12.6. Управление конфигурацией

Программная система имеет тенденцию *изменяться с течением времени*. Сложное ПО состоит из *многих частей* и разрабатывается *многими людьми*.

Если рассматривать эти три характеристики совместно, то можно увидеть серьезные проблемы:

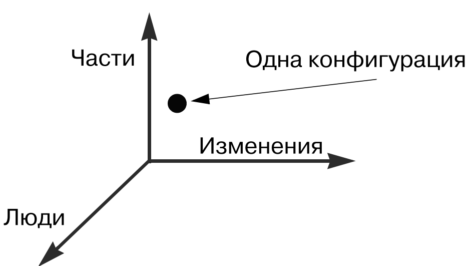


Рис. 12.12. Три измерения управления конфигурацией ПО

Оставьте только *два* измерения, и проблемы по-прежнему сохранятся. Фактически даже *одно* из них требует управления конфигурацией.

Разнообразие в управлении конфигурацией

Источником, который управлению конфигурацией и дал это имя, является задача по комбинированию или конфигурированию отдельных частей ПО в единую систему. Для простоты рассмотрим два измерения – «Части» и «Изменения». В случае программ частями являются модули, такие как методы, классы и кластеры. Каждый модуль проходит через последовательные версии, в соответствии со своим собственным расписанием и ограничениями; по аналогии с музыкальным произведением, это скорее контрапункт, чем гармония, скорее фуга Баха, чем военный марш. Но системе в целом нужно развиваться в своем ритме, имея собственную историю. Приходится отдельные куски склеивать вместе в их текущем состоянии и выпускать очередной релиз системы. На жаргоне скомпонованная система называется «билд», или **сборка** (build), и это источник, угрожающий бедствиями.

Так просто использовать версию 3.1 модуля A и версию 2.5 модуля B, в то время как версия модуля A прошла сертификацию на совместную работу с версией 2.4 модуля B. Многие катастрофические ошибки в работе ПО возникали по вине этой, казалось бы, тривиальной ошибки управления сборкой. Ошибки перестают быть тривиальными с ростом размера системы и времени ее существования.

Но управление конфигурацией не сводится только к проблемам сборки. Сложности могут возникать даже при разработке единственного модуля. Вот типичные вопросы:

- *Когда* модуль был последний раз модифицирован?
- *Кто* модифицировал модуль в период между сентябрем и декабрем прошлого года?
- Когда была *выпущена* версия n?
- *Что* изменилось в версии $n+1$ в сравнении с версией n?
- Что послужило *причиной* данного изменения?
- Эта ошибка все еще *существует* в текущей версии?
- Если нет, то когда она была *исправлена*?
- Можем ли мы *вернуться* к версии модуля M от 15 марта этого года?

Данный аспект управления конфигурацией называют **контролем версий**.

Большинство широко распространенных средств управления конфигурацией занимаются именно этими двумя вопросами – автоматической сборкой и контролем версий. Мы рассмотрим оба эти аспекта, а в следующем разделе обсудим проект глобального хранилища (репозитория), расширяющий управление конфигурацией до поддержки общей инфраструктуры проекта.

Инструменты сборки: от Make до автоматического анализа зависимостей

Источником вдохновения для построения инструментария сборки послужила команда Unix – Make, разработанная Стюартом Фельдманом в 1977 году.

Реконструкция системы по заданному описанию зависимостей между модулями основывается на использовании *файла сборки* – makefile, представляющего список входов в форме:

```
target: source1, source2 …
    command1
…
```

Рис. 12.13. Стюарт Фельдман (2006)

Это описание говорит, что цель *target* включает источники *source*, которые обычно задаются файлами. Всякий раз, когда один из источников изменяется, цель должна быть перестроена, для получения цели из источников нужно выполнить команду или группу команд command, указанную в файле сборки.

Выполнение

```
make target
```

приведет к реконструированию цели; если при этом один из источников сам окажется целью одной из зависимостей, то вначале он будет перестроен таким же образом. В этом процессе порядок следования зависимостей не имеет значения, поскольку процессор **make** выводит нужный порядок применения команд. Язык, применяемый для файлов сборки, отличается от императивных языков программирования, являясь дескриптивным языком, не задающим явно точный порядок применения команд.

Понятие зависимости включает время обновления. Операционная система записывает метки времени (время последней модификации) для каждого файла. Конфигуратор Make будет применять зависимость только тогда, когда метка времени одного или более источников является более поздней, чем метка файла цели.

Типичный файл сборки для построения программы, написанной на С, имеет вид:

```
program: main.o module1.o module2.o
    cc main.o module1.o module2.o
%.c: %.o
    cc $<
```

Для программ на С принято сохранять их в файлах с расширением *.c*, например *name.c*. Компилятор – команда **cc —** генерирует по исходному коду объектный код, записывая его в файл с расширением *.o* (*name.o*). Команда **cc** выполняет двойную работу: будучи линкером, она применяется к одному или нескольким объектным файлам, создавая новый модуль с удаленными перекрестными ссылками. В приведенном файле сборки указано, что наша программа *program* должна быть собрана из трех объектных модулей (*main* – это главный программный модуль); для генерирования программы нужно применить к ним компилятор **cc**. Для описания того, как из исходного модуля получать объектный, можно было бы использовать три зависимости в форме

```
main.o: main.c
    cc main.c
```

Аналогично можно было бы записать зависимости для *module1* и *module2*. В файле сборки все три зависимости объединены в одно общее правило, применимое к любому *.o*-файлу; символ % является держателем места — вместо него может быть подставлено любое имя. Команда в строчке $< означает, что она должна быть применена к результату, полученному в предыдущей строке файла сборки (конечно, нотация не слишком прозрачная, но это простительно, особенно если учесть, что с 1977 года Make и его потомки помогли миллионам программистам правильно конфигурировать свои программы).

Команда **make** *program* будет делать то, что и ожидалось: вначале скомпилируются три исходных модуля и будут созданы соответствующие *.o*-файлы; затем они будут скомпонованы (снова той же командой **cc**), создав в результате *program*. Заметьте, Make автоматически определяет порядок этих операций, анализируя зависимости файла сборки.

Эти концепции применимы не только к программированию. Например, файл сборки можно было бы применить для генерации документации, где зависимости включали бы исходные файлы для документов в разных форматах (Microsoft Word, Open Office, TeX, Frame Maker), а команды позволяли бы преобразовывать эти документы в единый формат HTML или PDF.

Без всякой посторонней помощи Make устанавливает дисциплину управления сборкой. Он является примером успешного решения инженерных задач. Его главным ограничением является необходимость явного задания зависимостей. При изменении модулей могут меняться и зависимости, поэтому постоянно нужно быть уверенным, что файл сборки был корректно обновлен. Будучи программным продуктом, он должен проектироваться и сопровождаться так же, как и остальные компоненты ПО. Возможны некоторые его улучшения, подобные рассмотренным параметризованным правилам, но процесс формирования файла остается утомительным.

Более современный подход состоит в том, чтобы поставлять документируемую программу, содержащую всю необходимую информацию для автоматического построения зависимостей. Это то, что делает, в частности, EiffelStudio, — здесь нет аналога файла сборки, поскольку компилятор сам может определить, когда метод или класс был изменен, какие классы от него зависят и, следовательно, должны быть перекомпилированы.

Контроль версий

Инструменты, отвечающие за контроль версий, помогают сохранить трассировку успешных версий отдельного модуля. В нашей трехмерной картинке контроль версий соответствует горизонтальной плоскости, а в частном случае одного разработчика — горизонтальной оси координат.

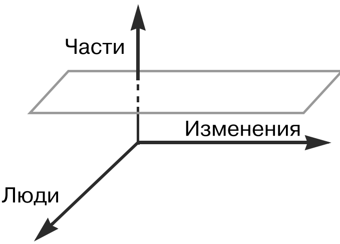


Рис. 12.14. Три измерения управления конфигурацией ПО

Части (модули в случае программ) подвергаются последовательным изменениям. Мы не можем позволить разработчикам редактировать их по своему усмотрению, а затем перекомпилировать всю систему. Это привело бы к хаосу. Вместо этого каждое изменение должно быть записано — *кто*, *когда*, *что* и *почему*. Необходимо уметь сравнивать последовательные версии и уметь откатываться к предыдущей версии.

Существует достаточно много средств контроля версий, коммерческих и свободно распространяемых. Некоторые из них — достаточно сложные интегрированные системы, хотя наиболее успешные являются простыми средствами, сфокусированными на основных проблемах, которые позволяют интегрировать их без особых усилий в процесс разработки ПО. Наибольшую известность получила линейка инструментов, название первого из которых было четырехбуквенным акронимом, а остальные стали трехбуквенными. Марк Рочкинд из Bell Labs разработал в 1972 году инструмент SCCS (Source Code Control System). Он был усовершенствован Уолтером Тичи в 1982 году под названием RCS (R — от Revision).

Рис. 12.15. Уолтер Тичи (2006)

Позже появился CVS (1986, C — от Concurrent, основанный на RCS), последняя версия — это SVN, новая реализация CVS.

Установка такой системы включает:

- *репозиторий* (хранилище), который содержит официальные успешные версии каждой части, прошедшей контроль версий.

  Концептуально репозиторий представляет базу данных, хотя обычно системы контроля версий не используют технологию баз данных;

- локальные копии частей, которые пользователи (разработчики ПО) могут хранить и модифицировать исходя из своих собственных потребностей.

Репозиторий хранится на сервере, и пользователи обычно получают доступ через сеть. Две фундаментальные операции доступны для пользователей:

- **Обновить (Update)** – создать локальную копию части, хранящейся в репозитории. По умолчанию выдается последняя версия части, но можно получить и любую предыдущую версию;
- **Зафиксировать (Commit)** – ввести часть в репозиторий, возможно, новую, но чаще модифицированную версию существующей части.

Фиксация создает новую версию, снабжая ее идентификатором версии. Общепринято идентификаторы версий задавать в виде последовательности чисел, разделяемых точкой.

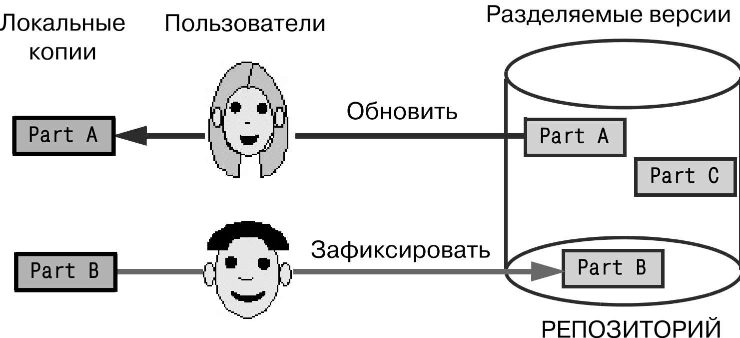


Рис. 12.16. Входной и выходной контроль

Так, на момент написания этого текста версия EiffelStudio имела номер 6.3, где 6 – это главный номер, который изменяется только для версий, вносящих принципиальные изменения, 3 – это младший номер, изменяющийся при каждом новом выпуске системы. Промежуточные версии, которые вносят, например, заплатки (patch), устраняющие ошибки, будут нумероваться как 6.3.m или даже 6.3.m.n.

Концептуально репозиторий сохраняет все версии. Кажется, что эта цель недостижима, поскольку требует чрезмерно большой памяти, Реалистичной эту технологию делает введение понятия «*diff*» — *различие*, название, пришедшее от команды Unix, которая для двух переданных ей файлов показывает их различие — строки добавлены, строки удалены, строки изменились. Если вы когда-либо, будучи на странице *History* в Википедии, выбирали «Compare selected versions» («Сравнить выбранные версии»), то могли видеть подобную картину сличения различий:

Представление «*diffs*» двух версий файла, скажем, $file_n$ и $file_{n-1}$ предназначено для человеческого восприятия, но *diff*-алгоритм может также вырабатывать специальную форму *d*, которая позволяет сопутствующему алгоритму реконструировать $file_n$, зная *d* и $file_{n-1}$, или получить $file_{n-1}$ по $file_n$ и *d*.

> Сопутствующий алгоритм прямолинеен: *d* описывает последовательность строк, добавляемых, удаляемых, изменяемых, поэтому достаточно применять эти операции к файлу в заданном порядке. Алгоритм diff не столь прост.

Как следствие такого подхода, в репозитории системы контроля версий нужно хранить только оригинальную версию каждого файла и последовательность его изменений. Когда приходит запрос на очередную версию, она реконструируется. Чаще в репозитории хранится последняя версия, поскольку в этом случае для наиболее часто возникающего запроса нет необходимости в реконструкции версии. В любом случае, поскольку «diffs», как правило, много меньше полной версии, такая технология позволяет сохранять полную историю каждого файла, иногда продолжающуюся в течение десятилетий и помнящую тысячи исправлений.

Как показано на примере Википедии, контроль версий полезен не только для программных модулей. Например, в нашей группе в ETH многие преподаватели работают над слайдами больших курсов, – все слайды хранятся в репозитории.

В случае разработки ПО контроль версий применяется не только к программным модулям, но и ко всем другим документам программного проекта, начиная от документа требований, продолжая документами проектирования и заканчивая результатами тестирования. Контроль версий прост в использовании и предотвращает многие неприятности. Следующее правило является одним из наиболее важных правил, которому необходимо следовать в процессе разработки ПО.

Почувствуй методологию

Используй контроль версий
Сохраняйте весь код и все документы программного проекта под управлением системы контроля версий.
При фиксации версии всегда записывайте причину и природу изменений.

Вторая часть совета связана с возможностью при фиксации новой версии ввести сообщение, которое будет храниться вместе с изменениями. Можно этого не делать, но так поступать не следует. Ситуация аналогична заданию заголовочного комментария для метода. Систематическое применение этого правила приводит к тому, что со временем репозиторий превращается в обогащенную базу знаний, хранящую эволюцию ПО.

Рассмотренный сценарий контроля версий прекрасно работает для случая единственного разработчика модуля. Более сложная ситуация возникает, когда несколько человек работают над одним и тем же модулем, например, классом. Конечно, на время обновления каждый разработчик может закрывать модуль для других пользователей, но обычно это слишком строгое требование. Когда вы фиксируете свои изменения после фиксации изменений другого разработчика, система контроля версий обнаружит конфликт и попросит вас разрешить его, предоставляя вам в помощь файл различий. В большинстве случаев конфликт легко разрешается, поскольку разные разработчики работают над разными классами или разными методами одного класса, но если возникает реальный конфликт, когда изменения пересекаются, то его разрешение требует внимательного рассмотрения. Не следует накапливать изменения, поскольку это может приводить к сложностям разрешения конфликтов.

Почувствуй методологию

Чаще фиксируйте изменения
Каждое важное изменение следует фиксировать. Это позволяет минимизировать конфликты и облегчает их разрешение.

Дополняющая часть совета включает «ветвление».

Почувствуй методологию

Ветвление
Не создавайте новую ветвь контроля версий, если только не собираетесь на старой основе разработать новый, независимо сопровождаемый продукт.

В системе контроля версий доступно ветвление, позволяющее разделить продукт на два, каждый со своей собственной системой идентификации. После ветвления каждая ветвь начинает жить собственной жизнью, так что через некоторое время их объединение становится трудной задачей.

Соблазн ветвления возникает достаточно часто, когда несколько разработчиков, работая над одним и тем же кодом, развивают его в разных направлениях, – понятно их желание работать независимо, оставляя проблемы воссоединения на будущее. Мудрость, накопленная сообществом разработчиков за многие годы, говорит, что подобная независимость – плохая идея. Небольшие хлопоты по разрешению небольших ежедневных конфликтов предпочтительнее «большого взрыва», когда собираются все разработчики, предлагающие результаты работы за несколько месяцев упорного труда, и у каждого из них все работает хорошо, но отказывается работать в совместном варианте.

Единственный случай, когда допускается ветвление, – это создание новой независимой линейки программного продукта.

Например, EiffelStudio имеет исследовательскую версию EVE (Eiffel Verification Environment), которая является ветвью версии 6.2. В данном случае обе ветви все еще остаются синхронизированными, подвергаясь регулярным реконструкциям, поскольку изменения имеют тенденцию воздействовать на разные части системы.

Управление конфигурацией, включающее управление сборкой, контроль версий, так же как и более продвинутые приложения, – это те «лучшие практики», отработанные современной инженерией программ, и их следует применять в каждом проекте – большом или малом.

12.7. Репозиторий проекта «в целом»

Управление сборкой, контроль версий поднимают специфические проблемы управления проектом. Дополняя эти частные решения, часто интегрируя их, в последние годы появились платформы «общего репозитория проекта», предоставляя общее хранилище проекту в целом. Одной из наиболее известных таких систем является *SystemForge*. Другим примером является разработанная в ETH система *Origo*.

Основная идея таких платформ состоит в обеспечении удобного и согласованного способа предоставления множества услуг, необходимых каждому проекту. Например, при создании проекта *Origo* для разработчиков автоматически создается репозиторий контроля версий, Web-сайт с раздельными правами для администратора, разработчиков и пользователей, электронный форум, Вики-страницы для документации проекта и многое другое.

Область применения таких средств не ограничивается программными проектами: любая деятельность, требующая сотрудничества, может воспользоваться предоставляемыми преимуществами.

12.8. Просмотр и документирование

Большие программные системы включают многие компоненты, связанные различными отношениями, такими как клиентские отношения и отношения наследования в ОО-мире. Методы в этом мире подвергаются многим перевоплощениям в своем долгом путешествии через поколения наследования – классы могут переопределять их, переименовывать, отменять их определение.

Средства просмотра помогают программистам разобраться в этом лабиринте. Они помогают получить ответ на типичные вопросы:

- Кто является родителями класса C? Его наследники, предки, потомки? Его клиенты, поставщики?
- Какой из предков впервые определил метод *f*?
- У какого предка я могу найти версию *f*, применяемую в классе C?

Связанной задачей является задача генерирования документации по тексту ПО. Возможно, вам потребуются версии класса на разных уровнях абстракции (контрактный облик или интерфейс класса), в разных форматах (HTML, PostScript). Насколько возможно, этот процесс должен быть автоматизирован и должен извлекать информацию из самого текста ПО, а не из различных документов, его сопровождающих. Если пользоваться внешней информацией, то всегда есть риск, что она устарела, не поспевая за изменениями ПО. Так как код может не содержать всю требуемую информацию, некоторые языки программирования предоставляют специальные конструкции для этих целей. Так, языки Java и C# позволяют задавать документируемые комментарии, обрабатываемые специальным инструментарием; в языке Eiffel для этих целей введено специальное предложение **note**, которое может и должно появляться в классах Eiffel, позволяя описать специальные свойства класса.

Средства документирования обрабатывают также и заголовочные комментарии методов, и по этой причине рекомендуется никогда их не опускать.

12.9. Метрики

Информатика не относится к естественным наукам: объекты ее изучения – это творения, созданные не природой, а человеком. Эти творения большие и сложные, так что к ним не всегда применимы эмпирические, количественные методы анализа, которые используют ученые для объектов реального мира. Специальные метрические инструменты применяются для программ.

Свойства, подлежащие измерению, включают атрибуты процесса, характеризующие усилия на разработку ПО, атрибуты созданного продукта, характеризующие код и другие результаты затраченных усилий. Измеряемые атрибуты процесса включают время разработки (глобальное, отдельными членами команды, время на разработку отдельных модулей), стоимость разработки, число и типы обнаруженных ошибок. Измеряемые атрибуты продукта включают:

- размер кода (используя хорошо разработанные метрики – число строк кода, размер генерируемого кода, число классов, методов, экспортируемых методов, процент кода, посвященного контрактам);
- покрытие требований (какой процент предусмотренной требованиями функциональности реализован в системе?);
- другие измерения функциональности, такие как число различных экранов, поддерживающих интерфейс конечного пользователя.

Метрические инструменты собирают такие данные. Они могут быть очень полезными, являясь частью общей политики обеспечения качества, которая использует результаты измерений для улучшения процесса разработки ПО.

12.10. Интегрированная среда разработки

Прогресс, достигнутый в разработке индивидуальных инструментов программной инженерии – компиляторов, интерпретаторов, редакторов, компоновщиков, средств управления

конфигурацией, – привел к объединению всех этих инструментов в интегрированную среду разработки – IDE (Integrated Development Environment). Интеграция означает, что для разработчика среда представляется единым инструментом, обычно интерактивным и графическим (поддерживается также возможность работы с командной строкой). Пользователи IDE могут выполнять в ней все задачи, связанные с разработкой ПО, или, по крайней мере, большинство таких задач. Вместо того чтобы готовить текст в редакторе, сохранять его в файле, потом переходить к новому инструменту – компилятору, затем к отладчику, – все это теперь можно выполнять в одном месте – в среде разработки.

Типичный графический интерфейс пользователя – GUI – все еще предоставляет разные подокна – редактора, компилятора, отладчика и так далее, но теперь все они связаны друг с другом. Их можно комбинировать для исследования различных свойств класса: видеть текст в окне редактора, структуру в окне документации, можно запустить на выполнение один из методов в окне отладчика. Можно обмениваться информацией между окнами, используя механизм перетаскивания – «drug and drop».

Одной из наиболее известных IDE является среда Eclipse, относящаяся к системам с открытым кодом и изначально ориентированная на Java. Из коммерческих сред следует отметить среду Microsoft Visual Studio.

Подобные среды включают инструменты, которые покрывают большинство задач, рассмотренных в этой главе: компиляцию, интерпретацию, редактирование, ввод и отображение графической информации, связывание и выполнение программ, их отладку, метрику. Некоторые среды включают и поддержку управления конфигурацией, но чаще всего предлагают интерфейсы к независимой системе управления конфигурацией.

12.11. IDE: EiffelStudio

В заключение обсуждения рассмотрим среду разработки EiffelStudio, как конкретный пример IDE, а также потому, что она поддерживает программистские концепции, изучаемые в этой книге.

> Заметьте, это описание концепций, а не руководство пользователя. Соответствующее приложение позволяет получить первые сведения, необходимые для работы в среде.

Реализация EiffelStudio использует свою собственную технологию; на момент написания среда включала примерно 2 миллиона строк Eiffel-кода (примерно 6000 классов) плюс поддержка С кода для исполняемой среды (runtime).

Общая структура

На следующем рисунке показаны главные компоненты EiffelStudio:

Центром среды – ее машиной – является EiffelStudio. Она обеспечивает пользователя ключевыми механизмами: просмотром и документацией, компиляцией, отладкой, взаимодействия между текстуальным и графическим (Diagram Tool) способом задания структуры, метрическим инструментарием.

Слева показаны несколько библиотек повторно используемых компонентов, ориентированных на разные области применения. Основными являются библиотека EiffelBase, покрывающая общие структуры данных и алгоритмы, и библиотека EiffelVision, которая предлагает графику, переносимую на разные платформы, такие как WIndows и Linux. Компоновщик

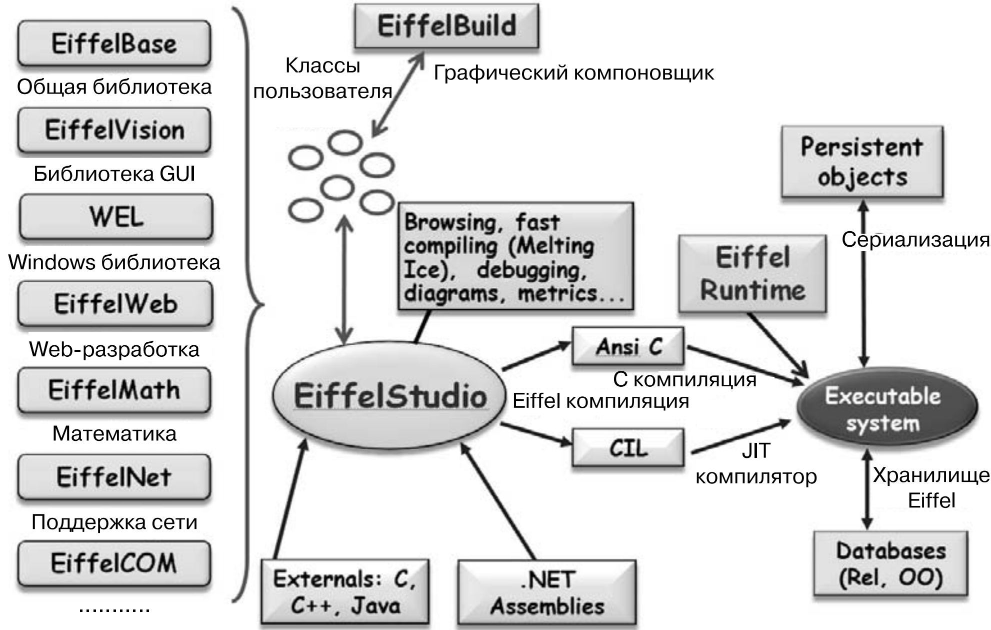


Рис. 12.17. Главные компоненты EiffelStudio

EiffelBuild, показанный в верхней части рисунка, позволяет строить графический интерфейс пользователя. Он генерирует код, вызывающий методы библиотеки EiffelVision, хотя можно непосредственно вызывать нужные методы, создавая интерфейс программным путем.

Нижняя часть рисунка отображает механизмы взаимодействия Eiffel с программами, написанными на разных языках, так же как и со сборками .Net, задающими так называемый управляемый код.

Рисунок показывает два механизма компиляции. Компилятор может сгенерировать C-код, используемый здесь в качестве переносимого ассемблерного языка, который на целевой машине компилируется в машинный код. Для платформы .Net компилятор создает байт-код для этой платформы, известный как промежуточный язык CIL (Common Intermediate Language), который затем JIT-компилятором транслируется в окончательный код. Для платформы .Net используется ее собственная исполняемая среда, но в схеме, базируемой на C, результат компиляции компонуется с собственной исполняемой средой EiffelStudio.

Созданная в результате работы выполняемая система (справа) может создавать сохраняемые структуры объектов, благодаря механизму сериализации, и обмениваться объектами с базами данных – реляционными и объектно-ориентированными.

Просмотр и документация

Возможности просмотра и документирования в EiffelStudio разработаны с особой тщательностью. Работа пользователя в студии характеризуется метафорой «выбрать и опустить»

(pick and drop) – выбираем «камешек», представляющий программный элемент, и опускаем его в «лунку» для выполнения подходящей операции над этим элементом. Более детально:

- выбираемыми элементами могут быть классы, методы, кластеры и другие элементы, такие как сообщение об ошибке;
- запускаем процесс «выбрать и опустить» щелчком правой кнопки мыши на любом представлении (имя, значок) выбранного элемента, появляющегося в любом месте;
- после захвата элемента курсор меняет свой вид, его изображение зависит от захваченного элемента: – для класса, – для метода;
- вы опускаете значок в лунку опять-таки щелчком правой кнопки мыши. Во многих случаях роль лунки может играть все окно или подокно. Например, если класс опустить в окно редактора, то в окне появится текст класса и станет возможным его редактирование;
- для удобства нет необходимости при перемещении камешка удерживать нажатой кнопку мыши, как это делается в распространенном механизме «перетащить и опустить». Один щелчок делается для захвата элемента и один для его размещения в нужном месте. Щелчок левой кнопки позволяет отменить операцию.

Множество механизмов документирования дополняют технологию «выбрать и опустить». Для любого класса или метода можно отобразить несколько обликов на различных уровнях абстракции, представляющих контракт, интерфейс, плоскую форму класса, так же как и списки клиентов, поставщиков, методов: для метода можно проследить его *историю* в предках и потомках.

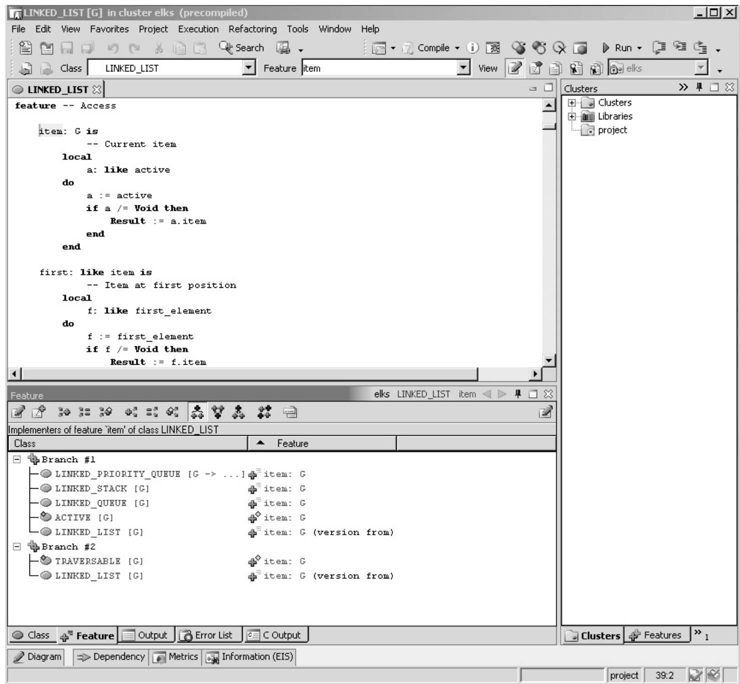


Рис. 12.18. Реализаторы метода

В качестве примера облика на снимке экрана, приведенном ниже, показаны «реализаторы» метода *item* из класса *LINKED_LIST* – все классы, в которых метод получал новую реализацию. Для получения приведенного результата, следуя технологии «выбрать и опустить», в верхнем окне было выбрано имя метода, затем оно было перетащено и опущено в нижнее окно, где был выбран облик «*implementers*», отображающий классы – реализаторы метода. Любой класс, метод или кластер, появляющийся на экране, может стать целью для технологии «выбрать и опустить».

Отображать такую информацию, как и любой другой облик, можно в разных форматах – HTML, PDF, RTF (Microsoft Word) и других форматах (достаточно просто добавить произвольный формат, написав для него соответствующий «фильтр»). При командной работе предоставляемые возможности облегчают работу, позволяя обмениваться информацией, начиная от наиболее детализированной формы – исходного кода, и заканчивая абстрактными представлениями – обликами и диаграммами.

Технология тающего льда

Технология компиляции в EiffelStudio комбинирует интерпретацию и компиляцию, чтобы обеспечить как быструю реакцию при внесении изменений в процессе отладки, так и высокую производительность скомпилированной системы.

Известно, что разработчик большой системы не компилирует ее каждый раз «с нуля». Никто не проводит месяцы, создавая код без его компиляции. Нормальной практикой является перекомпиляция системы, в которую внесены относительно небольшие изменения. Система может быть большой или малой, изменения могут большими или незначительными. Для небольших систем любой современный механизм компиляции будет достаточно быстрым. Критическим является случай *малых* изменений *большой* системы. Примером может считаться сама EiffelStudio с ее миллионами строк кода. Очередное изменение системы, ее расширение, можно реализовать в течение нескольких минут, понятно, что результаты хочется получить непосредственно, запустив систему и выполняя некоторый тест.

Это естественное желание разработчика задает ограничение на проектирование окружения.

Принцип тающего льда

> Время перезапуска системы после внесения в нее изменений должно зависеть от размера изменений, но не от размеров самой системы.

Технология компиляции вытекает из этого принципа. Метафора, выражающая этот принцип, отображена на рисунке. Думайте о скомпилированной системе как о глыбе льда. Изменения похожи на тающие капли воды, которые стекают с глыбы в результате тепла, порожденного вашими усилиями при работе с системой.

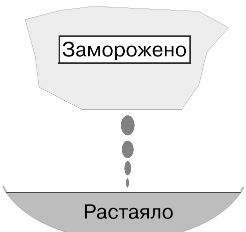


Рис. 12.19. Тающий лед

Таяние – это процесс внесения изменений в систему, замораживание – превращение системы в глыбу льда (помещение ее в холодильник) – это перекомпиляция системы.

Когда вы вносите изменения, «растаявшие» части по умолчанию не перекомпилируются, они интерпретируются – для них пишется байт-код. Но это относится только к измененным кусочкам кода, типично очень небольшому фрагменту системы. Как много можно изменить за несколько минут или за полчаса работы? Оставшаяся часть системы остается скомпилированной, поэтому общая производительность не пострадает из-за внесения интерпретируемого фрагмента. Общий механизм требует, чтобы компиляция и интерпретация сочетались друг с другом. Если скомпилированная подпрограмма вызывает подпрограмму, подвергшуюся модификации, то вызывается версия, использующая байт-код. Компилируемый код должен содержать переключатель, позволяющий вызывать правильную версию.

Вся EiffelStudio является открытым кодом, так что при желании понять детали этого тонкого устройства можно, просто проанализировав код.

Механизм таяния автоматический. Изменения могут быть сделаны либо путем редактирования текста в EiffelStudio, либо внешним инструментарием. При нажатии кнопки компиляции *Compile* механизм анализа зависимостей EiffelStudio находит минимальное множество элементов (это могут быть отдельные методы, а не обязательно классы), которые должны быть перекомпилированы. Сюда входят не только те элементы, которые вы явно изменили, но и другие, зависящие от них методы. Скрытие информации позволяет делать этот процесс более эффективным, поскольку, если метод изменился, но эти изменения не затронули его интерфейс, то клиентов класса перекомпилировать не нужно. Анализ не требует вмешательства пользователя, в частности, файла сборки или его эквивалента: как отмечалось ранее, семантической информации, присутствующей в тексте, вполне достаточно. Вычисление зависимостей в EiffelStudio выполняется автоматически. Это одно из преимуществ инструментов, встроенных в IDE и специально предназначенных для статически типизированного ОО-языка.

Со временем растаявшая часть (вода в нашей метафорической бутылке или более прозаически – файл, содержащий сгенерированный байт-код) становится большой, и тогда потери производительности могут стать заметными, так что потребуется новая заморозка. Попрежнему это возрастающая компиляция, компилируется не более того, что растаяло. Заморозка выполняется автоматически и в том случае, если изменениям подвергся внешний код, так как интерпретатор не может работать, например, с кодом на языке C.

Как таяние, так и заморозка являются автоматически выполняемыми операциями в режимах компиляции, предполагающих выполнение в EiffelStudio. Третий режим компиляции – финализация – позволяет сгенерировать код, полностью независимый от EiffelStudio. Финализация также выполняет интенсивную оптимизацию – удаление мертвых участков кода, статическое связывание (некий эквивалент динамического связывания, о котором пойдет речь в главе, посвященной наследованию).

Оптимизации возможны во всех режимах компиляции. Удаление мертвого кода основано на анализе, показывающем, что некоторые подпрограммы никогда не вызываются. Но в дальнейшем программист может добавить вызов удаленного метода. Это означает, что такая оптимизация не является возрастающей, она требует анализа всей системы и может быть выполнена полностью только когда компилятор генерирует полный код системы.

Финализация, благодаря прогрессу в технике компиляции и успехам современной аппаратуры, выполняется за разумное время. Если несколько лет назад на полную компиляцию

EiffelStudio уходило несколько часов, то теперь это время сокращено до 15 минут на обычном персональном компьютере.

В дополнение к трем рассмотренным режимам компиляции – таяние, заморозка и финализация – введен еще один режим – предкомпиляции, обрабатывающий библиотеки, такие как EiffelBase и EiffelVision, так что они могут быть присоединены без компиляции. Базисные библиотеки обычно используются в предкомпилированной версии или выполняют предкомпиляцию в момент инсталляции этих библиотек.

12.12. Ключевые концепции, введенные в этой главе

- Инструментарий ПО для программной инженерии играет ту же роль, что и средства CAD для проектировщиков и инженеров, работающих в архитектуре, электронике, машиностроении.
- Для реализации программ, написанных на языке высокого уровня, применяются два подхода: компиляция, преобразующая программу в исполняемый код, и интерпретация, которая предлагает машину, непосредственно выполняющую исходную программу.
- Некоторые реализации комбинируют компиляцию и интерпретацию. Например, компиляция может производить не машинный код, а код на промежуточном языке, который затем будет интерпретироваться. Примером смешанной стратегии компиляции и интерпретации является технология тающего льда, которая поддерживает возрастающую разработку путем интерпретации частей программы, подвергшихся изменению, оставляя остальную часть программы скомпилированной.
- Компиляторы выполняют несколько задач: лексический и синтаксический анализ, семантический анализ, генерацию кода и его оптимизацию. Традиционно эти задачи решались на нескольких последовательных «проходах» компилятора. Теперь компиляторы строят и декорируют базисные структуры данных – АСТ и таблицу идентификаторов.
- Компоновщик или линкер связывает скомпилированные модули, разрешая ссылки.
- Загрузчик загружает программу в память, готовя ее к выполнению.
- Исполняемая среда (runtime) обеспечивает поддержку в период выполнения программы.
- Редакторы и CASE-инструментарий помогает строить программу из текстового и графического ввода. При работе с графикой и текстом должно поддерживаться «циклическое обращение».
- Инструментарий компоновки, такой как Make, собирает систему из ее компонентов на основе описания зависимостей.
- Инструментарий контроля версий сохраняет последовательные версии частей программы, храня различия (diffs) между версиями.
- Средства просмотра и документирования облегчают анализ и понимание больших программных систем.
- Интегрированная среда разработки – IDE – поддерживает основные задачи разработки ПО, предоставляя взаимосвязанную коллекцию инструментов с согласованным интерфейсом.

Новый словарь

Branching	Ветвление	**Browsing**	Просмотр
Bytecode	Байт-код	**CASE**	Средства автоматизации программиста
Commit	Фиксация (транзакций)	**Debugger**	Отладчик
Configuration management	Управление конфигурацией	**Diff**	Инструментарий, определяющий различия
DSL	Проблемно-ориентированный язык	**IDE**	Интегрированная среда разработки
Jitter	JIT-компилятор	**Melting Ice**	Технология «тающий лед»
Jitting	Компиляция байт-кода	**Pass (of a compiler)**	Проход компилятора
Metric tool	Метрический инструментарий	**Runtime**	Исполняемая среда
Round-trip engineering	Циклическое обращение. Ограничение при взаимных преобразованиях текста и графики	**Update**	Обновление
		Version control	Контроль версий
		Virtual machine	Виртуальная машина

12-У. Упражнения

12-У.1. Словарь

Дайте точные определения терминам словаря.

12-У.2. Карта концепций

Добавьте новые термины в карту концепций, построенную в предыдущих главах.

12-У.3. Интерпретатор и компилятор

При рассмотрении концепций этой главы, связанных с абстрактным синтаксисом, предлагалось для небольших языков программирования написать соответствующий интерпретатор и компилятор; так как эффективное решение требует рекурсии и наследования, то соответствующие упражнения на эту тему появятся после изучения соответствующих разделов.

Часть III

Алгоритмы и структуры данных

Три главы третьей части посвящены фундаментальным алгоритмическим методам.
В первой главе дается обзор некоторых структур данных и связанных с ними алгоритмов, составляющих основу повседневной работы программиста. Здесь также рассматриваются приемы, позволяющие оценить эффективность алгоритмов. Еще одна, не менее важная тема связана с ОО-механизмом – универсальностью.
Во второй главе более глубоко рассматривается фундаментальная программистская техника, применяемая и при построении выводов, – рекурсия.
Концепции, рассмотренные в этих двух главах, используются в третьей главе для рассмотрения в полной мере инженерного решения интересной алгоритмической задачи – топологической сортировки – упорядочения элементов, удовлетворяющих множеству ограничений.

13

Фундаментальные структуры данных, универсальность и сложность алгоритмов

Бывают в жизни такие вечера, что кажется, за день ничего не удалось сделать, и день прошел в тасовании каких-то вещей с одного места на другое. Зачастую работа программ напоминает такую деятельность. Большинство из них, подобноTraffic с его списковыми структурами, задающими линии метро, проводят большую часть времени, размещая объекты в репозитории и занимаясь поиском ранее сохраненных объектов.

Такой репозиторий – хранилище элементов (items), называют **контейнером**. Список является одним из примеров контейнера. Есть много других видов контейнеров, которые различаются памятью, требуемой для хранения элементов, и скоростью выполнения операций (вставка, получение, удаление элемента, поиск элемента, удовлетворяющего некоторому условию, операция, применяемая ко всем элементам контейнера).

В этой главе мы изучим некоторые фундаментальные контейнерные структуры – массивы, разного вида списки, хэш-таблицы, стеки, очереди, – которые используются в самых разных областях приложения. В ходе этого рассмотрения нам предстоит осознать три важнейшие программистские концепции:

- роль *типов* в создании надежного ПО;

- *универсальность*: как объявлять классы контейнера, безопасные по типу хранимых элементов;
- *алгоритмическую сложность*, приемы оценивания производительности алгоритмов и структур данных.

По ходу рассмотрения мы встретимся с несколькими правилами хорошего стиля проектирования, такими как соглашение об именовании повторно используемых компонентов.

13.1. Статическая типизация и универсальность

Первая проблема, возникающая при работе с контейнером, – это проблема типизации.

Статическая типизация

Все сущности в наших программах объявляются с указанием типа. Это правило позволяет компилятору проверять, что любая операция, которую вы хотите применить к сущности x, например вызов метода $x.f(a)$, использует метод, разрешенный для применения. Компилятор может:

- найти объявление сущности x и установить тип T, заданный при ее объявлении;
- найти класс, задающий тип T;
- в этом классе проверить, что существует метод f, принимающий аргументы, число и тип которых соответствует аргументам в точке вызова.

Эта политика известна как **статическая типизация**: статическая, поскольку свойства типа специфицируются в тексте программы и могут быть проанализированы во время компиляции. Альтернативой является динамическая типизация, отказывающаяся от объявления типа и вынужденная ждать момента выполнения для обнаружения того факта, что метод не может быть применен к данной сущности. В предыдущих главах мы видели, что некоторые языки, как например, Smalltalk, предпочитают динамическую типизацию.

Выбор статической типизации, принятый в большинстве современных ОО-языков программирования, включая Java, C#, Eiffel, основан на двух основных аргументах:

- *ясность*: объявляя каждую сущность с точно определенным типом и каждый метод с точно определенной сигнатурой, мы задаем цели, стоящие перед нами, и облегчаем чтение и сопровождение программы;
- *надежность*: неверный вызов метода всегда является результатом программистской ошибки. Слишком большую цену приходится платить за позднее обнаружение ошибок в работающей программе, о чем уже шла речь ранее.

Статическая типизация для контейнерных классов

Как можно применить принципы статической типизации к контейнерам? Мы уже знакомы со списками, так как мы видели, что экземпляры *LINE* являются списками экземпляров класса *STATION* с методами, такими как

```
extend (s: STATION)                                    – Команда
         –Добавить s в конец линии
item: STATION                                          – Запрос
         – Станция в текущей позиции курсора)
```

Предположим теперь, что мы хотим иметь класс *LIST*, который может описывать списки *чего угодно*: список станций метро, список целых чисел, список объектов некоторого задан-

ного типа. Класс должен иметь вышеприведенные методы, но невозможно объявить тип *s* в методе *extend* или результат *item* без знания типа элементов списка: *STATION*, как выше, или любого другого типа, который вы выбрали для объектов конкретного списка.

Конечно, можно написать несколько классов: *LIST_OF_STATION*, *LIST_OF_INTEGER* и так далее. Не хотелось бы делать этого, так как тексты классов во многом были бы идентичными, за исключением некоторых объявлений. Такое дублирование или квази-дублирование противоречит принципам экономии и повторного использования.

Идея универсальности состоит в том, чтобы задавать один-единственный класс, но ***параметризованный***, так, чтобы он мог поддерживать разные типы без перепрограммирования.

Универсальные классы (классы с родовыми параметрами)

Используя универсальность, объявим класс *LIST* следующим образом:

```
class LIST [G] feature
    extend (s:G)
                  —Добавить s в конец списка.
          do … end
    item: G
          — Элемент в текущей позиции курсора.
    … Другие методы и инварианты…
end
```

G – это просто имя, известное как **формальный родовой параметр** (таких параметров у класса может быть несколько). Родовой параметр задает тип, так что его можно применить для объявлений внутри класса, как в нашем примере для аргумента *s* в методе *extend* и при объявлении результата запроса *item*.

Какой же тип обозначает *G*? Сам класс на этот вопрос не отвечает. Используя класс *LIST*, можно объявить, например:

```
first_1000_primes: LIST [INTEGER]
stations_visited_today: LIST [STATION]
```

Каждое такое объявление должно специфицировать тип, задав фактический родовой параметр – здесь *INTEGER* и *STATION* соответственно, указав тем самым, что обозначает *G* в данном конкретном случае.

Эта техника решает проблему статической типизации для общих контейнерных классов. Объявим переменные:

```
some_integer: INTEGER
some_station: STATION
```

Следующие операторы будут правильными:

```
first_1000_primes.extend (some_integer)
stations_visited_today.extend (some_station)
some_integer:= first_1000_primes.item
some_station:= stations_visited_today.item
```

Здесь все удовлетворяет правилам типа. Формальный аргумент *extend* в *LIST* имеет тип *G*; это значит *INTEGER* для *first_1000_primes*, объявленного как *LIST* [*INTEGER*], и *STATION* для *stations_visited_today,* поэтому вполне законно передать целое в качестве фактического аргумента в первом случае, и станцию метро – во-втором. То же справедливо и для результата *item*.

Но с другой стороны, следующие вызовы ошибочны:

```
first_1000_primes.extend (some_station)
stations_visited_today.extend (some_integer)
```

На этапе компиляции возникнут ошибки:

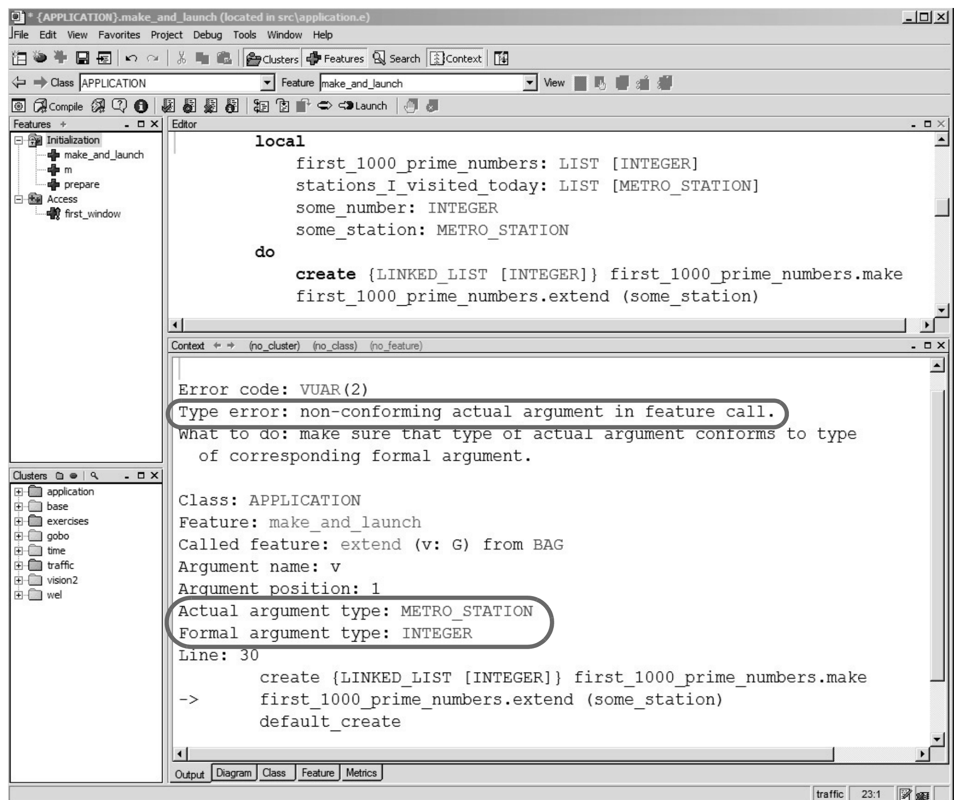


Рис. 13.1.

До середины 80-х годов на шоссе 101 от Сан-Франциско до Лос-Анджелеса и Сан-Диего водителей ждало только одно прерывание – светофор в Санта-Барбаре, создающий вечную пробку. Губернатор Калифорнии в семидесятых годах Джерри Браун ответил на жалобы недовольных в чисто калифорнийском стиле, что на самом деле они должны быть благодарны за возможность сделать паузу и расслабиться. Именно так и следует реагировать на ошибки, полученные в период компиляции.

Почувствуй методологию

Дзен и искусство реагирования на сообщения компилятора
Когда компилятор отвергает ваш класс, не надо браниться. Выдохните, налейте чашечку зеленого чая, задумайтесь о смысле жизни. Тогда вы осознаете, сколько часов отладки вам пришлось бы потратить, если бы эта ошибка не была обнаружена сейчас, а встретилась позже в процессе выполнения программы, возможно уже у заказчика. Задумайтесь, как избежать подобных ошибок в будущем, и радуйтесь жизни.

В рассмотренном нами случае система типов современного языка программирования защищает нас от ошибок, в частности, ее механизм универсальности, обеспечивающий разумное сочетание гибкости и безопасности.

Дальнейшее рассмотрение требует ввода новых понятий.

Определения: родовой или универсальный класс, родовое порождение

Универсальный (родовой) класс – это класс с одним или несколькими родовыми параметрами, задающими типы. Тип, полученный подстановкой фактических родовых параметров в универсальный класс, является **родовым порождением** этого класса.

Класс *LIST* является универсальным классом; тип *LIST* [*INTEGER*], полученный из *LIST* подстановкой родового параметра *INTEGER*, является родовым порождением *LIST*.

Все контейнерные классы, изучаемые в этой главе, такие как *ARRAY* [*G*], *LINKED_LIST* [*G*], *HASH_TABLE* [*G*, *H*], являются универсальными, родовыми классами. Родовой параметр с именем G всегда задает тип элементов в контейнере. Конечно, для родового параметра можно выбирать любое имя, лишь бы оно не совпадало с фактическим именем одного из классов системы.

Универсальность – это название механизма, позволяющего классам иметь родовые параметры и, как результат, допускать типы, полученные родовым порождением.

При рассмотрении методов класса обычно используется термин «формальные и фактические аргументы», оставляя термин «параметры» для универсальных классов. Это соглашение не универсально, и иногда термин «параметры» применяется и для аргументов методов класса. Тем не менее, следует осознавать разницу между этими двумя вариантами «параметризации».

Правильность против корректности

Целью механизма универсальности является, как отмечалось, проверка правильности некоторых видов программ (тех, что включают контейнерные структуры). Универсальность – это то, что делает «правильными» такие вызовы, как *first_1000_primes.extend (some_integer)*. Правильность означает, что вызовы удовлетворяют правилам типа языка, а, следовательно, компилятор их допускает.

Это, однако, еще не означает, что такие операторы всегда будут работать корректно. Цель вызова *first_1000_primes* может быть **void** во время выполнения, *extend* может иметь предусловие, которому *some_integer* не удовлетворяет. Следует различать два разных понятия.

Определения: правильность, корректность

Программа **правильна**, если она удовлетворяет всем правилам типа и другим согласованным статическим правилам, гарантирующим, что некоторые виды неверного срабатывания никогда не произойдут.
Правильная программа **корректна**, если она всегда выполняется в соответствии с желаемым поведением и никогда не станет причиной нарушения контракта или других сбоев в период выполнения, приводящих к отказам.

Определение корректности применимо только к правильным программам. Фактически, для *статически типизированного* языка (языка с точными правилами правильности, такого как Eiffel) нет смысла говорить о корректности, если программа не прошла проверку на «правильность».

Эти определения обобщают правило, утверждающее, что каждый уровень свойств языка имеет смысл при условии выполнения свойств предыдущего уровня. Мы уже встречались с этим утверждением, сформулированным для трехуровневого описания языка, к которому теперь можно добавить еще один уровень, расположив его между семантическим и синтаксическим уровнем. Точно так же, как синтаксические правила определены только для лексически корректных текстов, так и правильность, называемая также *статической семантикой,* определена только для синтаксически корректных текстов.

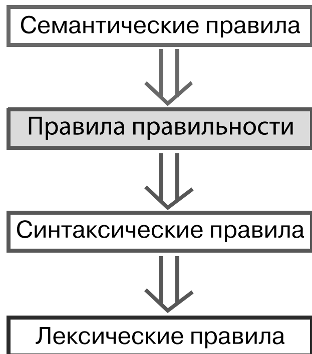


Рис. 13.2.

Примером «некоторого вида неверного срабатывания», устраняемого проверкой на правильность, является попытка вызова объектом метода, который не применим для обработки.

Почему вводятся два понятия? Не было бы проще, если бы «правильность» влекла «корректность»? Зная, что программа «прошла» компилятор, можно было бы спокойно отдыхать, будучи уверенным, что во время исполнения программа будет работать нужным образом. Это Мечта программиста. Несмотря на то, что языки программирования с введением статических правил стали лучше и теперь способны обнаруживать ряд ошибок во время компиляции, все еще остаются ситуации, приводящие к ошибкам, которые могут быть обнаружены только во время выполнения. Для них предназначены механизмы периода выполнения, такие как «обработка исключительных ситуаций».

Давним предметом поиска, «философским камнем» в программистских исследованиях является стремление приблизить правильность к корректности. Граница регулярно изменяется, сближая эти два понятия. Вероятно, одним из наиболее важных достижений последнего времени является возможность исключения *void-вызовов*, благодаря правилам типа, использующим механизм *присоединяемых типов*, который кратко был описан в предыдущих лекциях.

То, что раньше было источником серьезных и непредсказуемых ошибок в период выполнения, теперь становится предметом стандартной проверки компилятора. Это важное свидетельство успехов на пути, ведущему к *доказательству* корректности программы.

До тех пор, пока доказательство не станет обыденным делом, правильность и корректность будут отличаться. Тем не менее, статическая типизация, особенно если она сочетается с контрактным проектированием, дает инструмент, позволяющий справиться с «жучками» до того, как они вас «укусят».

Классы против типов

Универсальность позволяет нам лучше разобраться в отношениях между классами и типами.

Тип – это описание множества значений периода выполнения: тип *INTEGER* задает свойства целых, тип *STATION* задает свойства объектов (периода выполнения), представляющих станции.

Класс является программным модулем, определяющим коллекцию компонентов (полей и методов) и их свойств, таких как инварианты класса, применимых к множеству объектов периода выполнения.

Связь этих двух понятий очень тесная: множество объектов периода выполнения, ассоциированное с классом, является *типом*, если только класс не является универсальным. Любой класс, не являющийся универсальным, такой как *INTEGER* или *STATION*, на самом деле представляет тип и может использоваться в качестве такового при объявлении сущностей, как это делалось в ранее приводимых примерах:

```
some_integer: INTEGER
some_station: STATION
```

Классы используются двояко: как в роли базисных конструкций – модулей программы, являясь в этом случае статическим понятием, так и в роли механизма типизации объектов – динамическое понятие, центральное для ОО-программирования, которое лучше было бы называть КО-программированием (классо-ориентированным).

Связь между классами и типами остается такой же сильной и для универсальных классов. Новый поворот состоит в том, что универсальный класс, такой как *LIST* или *ARRAY*, более не задает тип – он задает шаблон типа, параметризованный тип. Для получения типа достаточно выполнить родовое порождение, задав фактические родовые параметры. Например:

- *INTEGER* и *STATION* являются классами, но они также являются и типами. Это справедливо для любого неуниверсального класса;
- *LIST* и *ARRAY* являются классами; *LIST* [*STATION*] и *ARRAY* [*INTEGER*] являются типами. Родовое порождение любого универсального класса является типом.

Дадим точное определение.

Определения: тип класса, универсально порожденный, базовый класс

Тип класса – это:

T1 класс, не являющийся универсальным;

T2 родовое порождение – оно, как говорилось, задается именем класса (называемым **базовым классом** типа), за которым следуют подходящие фактические родовые параметры. В этом случае говорят, что тип является **универсально порожденным**.

Вложенные родовые порождения

Осталось уточнить, чем может быть *фактический родовой параметр* для универсального класса. Ответ напрашивается: *типом*. Вы могли видеть это в последних примерах: в *LIST* [*STATION*] фактический родовой параметр *STATION* является типом, так же как и *INTEGER* в *ARRAY* [*INTEGER*].

Возможно, вы ощутили некоторую странность в этих определениях.

- В только что приведенном определении типов (предложение T2) говорится, что они могут быть получены из класса и фактических родовых параметров.
- Затем фактические параметры определяются как типы.

Не зацикливается ли это определение («масло масляное»)? Нет. Просто это пример рекурсивного определения, которое строит новые элементы из ранее определенных – в данном случае речь идет о построении множества типов. Рекурсивное определение должно иметь базовую, не рекурсивную часть определения. Процесс понятен:

- благодаря предложению T1 определения мы знаем, например, что *STATION* – не универсальный класс, является типом;
- тогда мы можем использовать T2, чтобы вывести, что *ARRAY* [*STATION*] также является типом.

Рекурсия – восхитительная техника, применяемая не только в подобных определениях, но и в программах и структурах данных. Мы посвятим ей отдельную главу, следующую за этой, но уже этого примера достаточно, чтобы показать, что ничего странного и несогласованного нет в рекурсивном определении «типа».

Определение открывает, фактически, интересные возможности. Тип, используемый в качестве фактического параметра в T2, не обязательно определяется предложением T1; он может определяться, в свою очередь, предложением T2 – другими словами, параметр может быть универсально порожденным. Это позволяет задавать такие типы, как

```
LIST [LIST [INTEGER]]
LIST [ARRAY [STATION]]
ARRAY [ARRAY [ARRAY [INTEGER]]]
```

Такая вложенность допускается без ограничений. Это не только приятная теоретическая возможность, но и практичный механизм, который появится при определении списка списков, списка массивов и других многоуровневых контейнеров.

13.2. Контейнерные операции

Во второй части этой главы дается обзор фундаментальных контейнерных структур, начиная с массивов, связных списков, других видов списков, и заканчивая стеками и хэш-таблица-

ми. Все они широко используются на практике. У всех у них много общего и есть своя специфика.

- Большинство базисных операций одни и те же: вставка и удаление элемента, поиск заданного элемента, определение числа элементов ...
- Каждый вариант контейнера по-своему реализует эти операции. Эти различия затрудняют обеспечение равно эффективной реализации для всех операций. Массивы позволяют получать элемент весьма быстро, если известен его индекс, но они медленны, если необходимо вставить новые элементы. Связные списки эффективны для вставки, но медленнее, чем массивы, для доступа по индексу. Другие структуры также имеют свои «за» и «против». Не существует единой структуры, оптимальной во всех ситуациях.

Когда возникает необходимость в контейнере, всякий раз приходится выбирать одну из доступных структур в зависимости от операций, которые необходимо выполнять над контейнером.

Прежде чем начать рассматривать специфические виды контейнеров, дадим обзор фундаментальных операций: вначале рассмотрим запросы, а затем команды. Пусть *G* будет обозначать тип элементов контейнера, и он будет всегда первым родовым параметром соответствующих классов, как в *ARRAY* [*G*] или *LINKED_LIST* [*G*].

Запросы

Одна из операций, необходимая всем контейнерам, – это запрос, определяющий, пуст ли контейнер (не содержит элементов). Запрос, возвращающий значение *BOOLEAN*, называется *is_empty*. Его сигнатура проста:

```
is_empty: BOOLEAN
```

Другими словами, у него нет аргументов, он вызывается в форме *c.is_empty*, возвращая булевское значение для каждого контейнера *c*.

Выяснить, находится ли некоторый элемент в контейнере, можно с помощью запроса

```
has(v:G):BOOLEAN
```

Для определения числа элементов в контейнере служит запрос:

```
count: INTEGER
```

Инвариант, применимый ко всем рассматриваемым контейнерным классам:

```
is_empty = (count = 0)
```

Для получения элемента контейнера, определяемого политикой контейнера, а не клиентом:

```
item: G
```

Некоторые контейнеры, такие как массивы, позволяют получить элемент по заданному клиентом индексу, как в предложении «дайте мне третий элемент». Такой запрос с параметром имеет вид:

```
item (i:INTEGER): G
```

Используется запрос с тем же именем, но неопределенности нет, поскольку различаются сигнатуры запросов.

Индекс, являющийся целым числом, является частным случаем **ключа** элемента, позволяющего получать элемент по ключу – информации, связанной с элементом. Есть много различных видов ключей – один из наиболее общих имеет тип **string**, как в контейнере, представляющем Web-страницу и позволяющем поисковой машине найти на странице заданный набор слов. Для строковых ключей запрос имеет вид:

```
item (i: STRING): G
```

Мы покажем, как обобщить тип ключа, допуская не только строки. Пока используемое имя запроса по-прежнему не приводит к конфликтам.

Команды

Процедура создания (конструктор класса) для контейнеров обычно называется *make*. Зачастую она не имеет аргументов. Но иногда задается аргумент, определяющий ожидаемое число элементов:

```
make(n:INTEGER)
```

Для всех контейнеров этой главы *n* является указанием на начальное создание структуры данных, но не является абсолютным максимумом.

Для наиболее общей операции по добавлению или замене элемента используется имя *put*. Эта операция применима с разными сигнатурами, соответствующими сигнатуре запроса *item*, но с добавлением еще одного аргумента, задающего новое значение:

```
put (v: G)
put (v: G; i: INTEGER)
put (v: G; k: STRING)
```

Постусловие всегда должно включать предложение

```
inserted: has (x)
```

и вдобавок должно выражать отношение с соответствующей версией *item*:

- *item* = *v*, если *put* не имеет аргументов (первый случай);
- *item* (*i*) = *v* для версии с целочисленным индексом;
- *item* (*k*) = *v* в последнем случае.

Процедура *put* может либо добавлять новый элемент, либо заменять существующий. Иногда эти два случая необходимо различать, применяя или:

```
extend (v: G)
extend (v: G; i: INTEGER)
extend (v: G; k: STRING)
```

с постусловием

```
one_more: count = old count + 1
```

или для замены использовать

```
replace (v: G)
replace (v: G; i: INTEGER)
replace (v: G; key: STRING)
```

с постусловием

```
same_count: count = old count
```

Когда существует либо *extend*, либо *replace*, то *put* обычно является синонимом одного из них, соответствуя (если оба присутствуют) более общему использованию. Во всех случаях постусловие *has*(*v*) выражает то, что после добавления элемента структура должна давать ответ «да», если делается запрос о его присутствии.

Процедура для удаления элемента в зависимости от контекста называется *remove* или *prune*.

Стандартизация имен методов для базисных операций

Имена, применяемые выше – *item, has, put* ..., – повторяются во всех библиотеках. Даже беглый взгляд на контейнерные классы показывает, что большинство из них содержит методы с этими именами.

Это осознанный выбор. Конечно, можно было бы придумывать новые имена для каждого класса, отражающие специфические свойства соответствующего контейнера. Но эти особенности уже отражены в сигнатуре, заголовочных комментариях и контрактах методов, например, для *put* в классе *ARRAY:*

```
put (v: like item; i: INTEGER)
        -- Заменить i-й элемент на v, если индекс в допустимом интервале.
    require
        valid_key: valid_index (i)
    ensure
        replaced: item (i) = v
```

Аналогично для *put* в классе *STACK:*

```
put (v: G)
        -- Поместить v в вершину стека.
    require
        extendible: extendible
    ensure
        pushed: item = v
```

Поэтому из-за сходства имен двусмысленность не возникает. Использование согласованных имен облегчает использование библиотек и – для новичков – обучение использованию:

при знакомстве с новым классом читатели могут быстро идентифицировать ключевые методы и их назначение.

Почувствуй методологию

Принцип стандартных имен методов
Используйте стандартные имена, когда они применимы, для методов ваших собственных классов, что улучшает согласованность и читабельность.

Автоматическая перестройка размера

Мы уже видели, что процедуры создания (обычно с именем *make*) позволяют задавать размер контейнера, но задаваемый аргумент следует рассматривать как указание начального размера, а не его постоянную максимальную границу. Структуры данных библиотеки EiffelBase почти всегда неограниченны или, имея начальную границу, могут изменять свой размер. Один из признаков хорошего программиста состоит в том, что в своих программах он избегает задания абсолютных границ.

Не позволяйте никому закрывать вас в камерах – это же справедливо и по отношению к пользователям вашей программы. У компьютеров большая память. Проектируя структуры данных, всегда возможно сделать так, что если размер данных превзошел ожидания, то нужно не отказываться работать с такой структурой, а перестроить ее, предоставив ей больше памяти. Если не следовать этому совету, можно столкнуться с самыми тяжелыми последствиями во время выполнения программы. Самое печальное, что программа прекратит работу, в то время как в системе достаточно пространства для ее работы.

Даже наши массивы должны быть перестраиваемыми.

Так случилось, что во время написания этой главы во всемирных новостях появилось сообщение о проблемах исследовательской экспедиции на Марс, организованной NASA и состоящей из двух спутников – Spirit и Opportunity. Первый из них замолчал более чем на день, не отзываясь на все попытки оживить его. Ошибка оказалась программной – для обработчиков файлов была выделена память фиксированного размера, а файлов оказалось больше, чем предполагалось.

Годом позже стала известна информация об отказе ПО, обеспечивающего выборы в Сан-Франциско. Причина была «в жестко зашитой константе, задающей максимальное число выборщиков, установленной слишком низкой».

Не попадайте в такие ловушки!

Почувствуй методологию

Не замыкайте в камеры ваших пользователей
Не используйте встроенные постоянные границы. Позволяйте вашим структурам данных перестраивать размер, адаптируясь к потребностям, возникающим при решении конкретной задачи.

Мы увидим, что для некоторых вариантов контейнера, особенно для массивов, перестройка является дорогостоящей по времени операцией, так что к ней следует относиться внимательно. Однако помните, что соображения эффективности никогда не могут служить поводом для отказа от перестройки границ структур данных, если в этом возникает необходи-

мость. Более того, структуры с фиксированными границами чаще всего вредят рациональному использованию памяти, так как «на всякий случай» запрашивают больше памяти, чем требуется в конкретной ситуации. Лучше вначале отвести память, исходя из ожидаемых средних потребностей, и перестроить структуру динамически, если необходимо.

13.3. Оценка алгоритмической сложности

Последний комментарий поднимает проблему *эффективности* (*производительности*), которая включает как время выполнения, так и требуемый объем памяти. Главная причина использования различных видов контейнеров состоит в том, что они обладают разной эффективностью по памяти и времени, зависящей от выполняемых над контейнером операций.

Необходим надежный способ сравнения производительности для выбора нужного типа контейнера. Недостаточно провести эксперименты над конкретным контейнером и сделать вывод, что «в среднем запрос на вызов элемента требует 10 наносекунд при работе с массивом и 40 наносекунд при работе со связным списком».

- Корректные рассуждения «о средних» требуют знания статистического распределения. Для размеров контейнеров трудно ожидать разумного определения такого распределения (сколько контейнеров может быть с 10 элементами, сколько с 1000 элементами и т. д.).
- Нельзя полагаться на результаты измерений из-за проблем масштабирования. Некоторые методы прекрасно работают для небольших размеров структуры, но становятся критическими для больших структур.
- Результаты измерений существенно зависят от окружения – компьютера, операционной системы, даже от языка программирования и компилятора; те же эксперименты в другом окружении могут дать принципиально различные результаты.

Основной способ оценки сложности алгоритмов свободен от таких привходящих обстоятельств. Он известен как **абстрактная сложность**, а также как *асимптотическая* сложность или нотация «О-большое».

Измерение порядка величин

Абстрактная сложность основывается на двух принципах.

- Рассматривает меру сложности как функцию от *размера* структуры данных. Для большинства примеров этой главы размер задается одним параметром – *count* – числом элементов контейнера.
- Функция определяется не точной формулой, а порядком величины, задаваемой нотацией «О-большое», как O(*count*).

Когда мы говорим, что время поиска элемента в списке из *count* элементов составляет O(*count*), это означает, что с ростом *count* оно возрастает, в худшем случае, **пропорционально** *count*. Другая операция может иметь время $O(count^2)$, означающее, что время возрастает самое большее пропорционально квадрату числа элементов. Те же соглашения действуют при оценке требуемой памяти.

Для такой меры:

- константные мультипликативные множители не играют значения: $O(100* count^2)$ означает то же самое, что и $O(count^2)$. Объяснение этого соглашения состоит в том, что не следует придавать большого значения умножению на константу, так как реализация того же алгоритма может быть в сто раз быстрее или медленнее, будучи перенесенной на другую машину. Но рост времени вычислений при изменении *count* не зависит от таких технических деталей;

- понятно, что не имеет значения и константная аддитивная составляющая: O($count^2$ + 1000) означает то же самое, что и O($count^2$). Конечно, константа может оказывать влияние при небольших значениях *count*, но с его ростом влияние становится ничтожным;
- аналогично: любая аддитивная составляющая с меньшим значением экспоненты также не оказывает влияния на порядок величины – O($count^3$ + $count^2$) означает то же самое, что и O($count^3$).

Как следствие, чтобы выразить тот факт, что алгоритм работает константное время, более точно – что на любой платформе время выполнения ограничено константой, будем говорить, что время работы O(1). Конечно с тем же успехом можно писать O(37) или O(1000), но принято писать O(1).

Математические основы

Нотация «O-большое» может показаться неформальной, но ее можно строго определить, как отношение между функциями.

Определение: нотация «O-большое» и абстрактная сложность

Пусть *f* и *g* – две *функции* над натуральными числами, задающие отображение в положительные вещественные числа. Говорят, что *f* есть O (*g*) – или, в более общем виде, *f* (*n*) является O (*g* (*n*)), указывая аргумент, – если существует константа *K*, такая, что *f* (*n*) / *g* (*n*)< *K* для каждого натурального числа *n*.

Алгоритм является O (*g* (*n*)) по времени или по памяти, если функция, задающая время выполнения или требуемый объем памяти при входном размере *n*, есть O (*g* (*n*)).

При анализе сложности алгоритма можно встретиться с таким выражением, как «*f* (*n*) = *g* (*n*) + O (n^2)», что является сокращением записи «*f* (*n*) = *g* (*n*) + *s* (*n*) для некоторой функции *s*, где *s* (*n*) есть O (n^2)». Тем самым устанавливается, что *f* «подобна» *g*, *отличаясь на терм* O (n^2).

Следствием определения является тот факт, что если функция есть **O** (n^2), то верно, что она есть также **O** (n^3), **O** (n^4) и так далее. Это потому, что O-большое задает верхнюю границу, а не представляет точную оценку. Полезные утверждения, связанные со сложностью, часто имеют вид: «Доказано, что сложность данного алгоритма есть **O** ($n^{2.5}$). Можно ли улучшить оценку и доказать, что его сложность есть **O** (n^2)?»

В теории сложности алгоритмов применяется нотация Тета-большое (θ (*g* (*n*))) в тех случаях, когда функция *g* асимптотически обеспечивает как нижнюю, так и верхнюю границу с разными константными мультипликативными множителями. Для простоты мы будем использовать нотацию O-большое, предполагая при этом, что функции *g* на данном этапе рассмотрения являются наилучшими из известных, характеризуя поведение алгоритма в наихудших условиях.

При анализе сложности алгоритмов часто возникают *логарифмы*. Например, лучшие алгоритмы *сортировки* списка из *n* элементов имеют сложность **O** ($n \times (\log n)$). В этой формуле не указывается основание системы логарифмов (2 или 10), поскольку изменение основания приводит к появлению мультипликативной константы $\log_b n = \log_b a \times \log_a n$.

Лучшее использование выигрыша в лотерею

Соглашение об игнорировании мультипликативной константы на первый взгляд кажется удивительным. Алгоритм, работающий $count^2$ наносекунд, считается хуже алгоритма, работающего $10^6 * count$ наносекунд, хотя последний работает быстрее при *count* меньше миллиона.

Следующее наблюдение позволяет понять преимущества алгоритмов, лучших по сложности. Рассмотрим четыре алгоритма, каждый из которых, непрерывно работая на вашем компьютере, за 24 часа может решить задачу максимальной размерности соответственно N_1, N_2, N_3, N_4. Предположим теперь, что алгоритмическая сложность наших алгоритмов соответственно составляет $O(n)$, $O(n \log n)$, $O(n^2)$, $O(2^n)$. Предположим еще (что менее вероятно), что вы выиграли в лотерею большую сумму и можете купить новый компьютер, работающий в 1000 раз быстрее старого. Что это может вам дать?

Для первого алгоритма со сложностью $O(n)$ теперь можно будет решить задачу в 1000 раз большего размера – $1000 * N_1$.

Для второго алгоритма со сложностью $O(n \log n)$ теперь можно будет решить задачу большего размера с множителем, близким к 1000.

Для третьего алгоритма со сложностью $O(n^2)$ теперь можно будет решить задачу большего размера с существенно меньшим множителем, равным квадратному корню из 1000, примерно равным 32.

Для четвертого алгоритма увеличение размера почти неощутимо, размер задачи увеличится на 10 (не в 10 раз, а на 10!).

Правильный вопрос, который нужно задавать при анализе сложности алгоритма, – не «каково время решения задачи», а «насколько большую задачу можно решить данным алгоритмом за фиксированное время при увеличении скорости работы компьютера».

> Рассмотрим в качестве примера программу, дающую ежедневный прогноз погоды. Благодаря компьютерному моделированию метеорологи добились серьезных успехов за последние два десятилетия. Прогноз дается по данным, собранным в различных точках географической сетки. Чем больше точек сетки учитывается, тем точнее прогноз. Рассматривая эффективность программы прогноза, следует в качестве критерия брать не время расчета прогноза для фиксированного множества точек (точный прогноз на следующий день бесполезен, если он считается 48 часов), а число точек сетки, которые можно учесть при фиксированном времени расчета, скажем, за час.

Абстрактная сложность дает нам взгляд на эффективность алгоритма, свободный от технических деталей, и позволяет оценить преимущества его потенциального улучшения.

Абстрактная сложность на практике

Говоря об абстрактной сложности, чаще всего рассматривают три разных варианта, особенности каждого из которых следует четко понимать.

Средняя сложность, ожидаемое среднее время или требуемый в среднем объем памяти алгоритма. Как отмечалось ранее, говорить о среднем можно только в предположении о существовании случайного распределения для входов программы. Обычно среднюю сложность считают в предположении, что все входы равновероятны (например, при сортировке массива из *n* элементов можно считать, что все *n*! возможных упорядочений элементов могут появляться с равной вероятностью).

Максимальная сложность, также называемая *сложностью в худшем случае*, дающая время или память для случая, на котором алгоритм работает дольше всего (требует максимальной памяти).

Минимальная сложность, также называемая *сложностью в лучшем случае*, дающая время или память для случая, на котором алгоритм работает быстрее всего (требует минимальной памяти). Этот критерий используется редко – его любят те, кто верит в удачу.

Презентация структур данных

В оставшейся части главы будем рассматривать фундаментальные структуры данных. Их презентация основана на библиотеке EiffelBase, содержащей повторно используемые классы для всех изучаемых понятий: *ARRAY*, *LINKED_LIST*, *HASH_TABLE*, *STACK* и так далее.

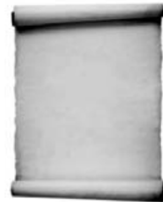
Описание дается в ориентации на клиента-программиста, того, кто будет использовать библиотечные классы в собственном приложении. По этой причине методы будут вводиться в их контрактном облике. Презентация объясняет базисные способы реализации; как правило, сама реализация не дается, но с ней при желании можно познакомиться, поскольку EiffelBase является библиотекой с открытым кодом, одна из целей которой состоит в предоставлении надежных образцов ОО-стиля программирования, проверенных в течение многих лет использования.

13.4. Массивы

Начнем с самого распространенного вида контейнера – массива.

Понятие массива является программистским понятием, но его важность определяется свойствами главной памяти компьютера, которая известна как RAM (Random Access Memory) – память со случайным доступом. Несмотря на такое имя, вовсе не предполагается, что в момент доступа к ячейке памяти компьютер случайным образом выбирает, из какой ячейки выбрать (или куда записать) значение (идея, сама по себе интересная). На самом деле эта память предполагает, что время доступа к ее ячейкам – для чтения или модификации – не зависит от адреса ячейки (понятие «случайный» следует трактовать, как во фразе: «*вы можете случайным образом выбрать ячейку и не беспокоиться о времени доступа*»). Если у Вас память в 2 GB, то доступ к первой ячейке (адрес 0) или к последней занимает одно и то же время.

Память RAM противопоставляется памяти с *последовательным доступом*, где, прежде чем получить доступ к элементу, необходимо пройти некоторый путь от начальной точки через предшествующие элементы к искомой точке. Магнитные ленты являются типичным примером: прежде чем головка чтения-записи будет установлена на нужном элементе, необходимо ленту перемотать от исходной позиции головки к нужной точке ленты. Аналогии встречаются и в устройствах, не связанных с компьютерами:



Последовательный

Случайный

Рис. 13.3. Последовательный и случайный доступ

Свиток слева позволяет последовательное чтение (и запись). Справа показаны почтовые ящики, с прямым к ним доступом.

Массивы используют все преимущества прямого доступа, позволяя работать со структурой данных, хранимой в непрерывном сегменте памяти; доступ к каждому элементу массива возможен по индексу (номеру) элемента:

Рис. 13.4.

Границы и индексы

Массив имеет нижнюю и верхнюю границы, задаваемые запросами класса *ARRAY*[*G*]:

```
lower: INTEGER
        -- Минимальный индекс.
upper: INTEGER
        -- Максимальный индекс.
```

Инвариант класса устанавливает, что *count* – число элементов (также известное как емкость) задается соотношением *upper* – *lower* + 1. Так как *count* >= 0, требуется выполнение условия:

```
lower <= upper + 1
```

Случай *lower* = *upper* соответствует массиву с одним элементом, *lower* = *upper* +1 соответствует пустому массиву (эти наблюдения можно визуализировать, передвигая на последнем рисунке вправо нижнюю границу или влево верхнюю, пока они не пересекутся). Это законные состояния массива.

Почувствуй методологию

Принцип экстремальных случаев

При проектировании структуры объектов, например контейнеров, рассматривайте экстремальные случаи – пустую структуру, структуру с одним элементом, «полную» структуру, если задана максимальная емкость, – и убедитесь, что определения имеют смысл и в этих случаях.

Известна длинная вереница «жучков», связанная с неадекватной обработкой экстремальных случаев. «Нормальное» мышление предполагает, что структура имеет элементы, но в процессе выполнения «вдруг» возникает экстремальная ситуация и все рушится. Приведенный выше методологический совет позволяет избежать подобных неприятностей.

Инвариант класса является первичным руководством для проверки того, что определение все еще имеет смысл. Здесь случай *lower* = *upper* +1 остается совместимым с инвариан-

том класса *lower* <= *upper* +1, полученное при этом минимальное значение *count* (*upper* -*lower* +1) все еще удовлетворяет требованиям.

Для чтения и модификации элемента массива необходимо указать его целочисленный индекс. Доступен запрос, определяющий корректность значения индекса:

```
valid_index (i: INTEGER): BOOLEAN
        - Является ли i правильным индексом, лежащим внутри границ?
    ensure
        Result implies ((i >= lower) and (i <= upper))
```

Создание массива

При создании массива необходимо указать предполагаемые значения границ:

```
your_array: ARRAY [SOME_TYPE]
…
create your_array.make (your_lower_bound, your_upper_bound)
При этом используется процедура создания:
make (min_index, max_index: INTEGER)
 - Выделить память массиву; установить интервал для индекса min_index ..
 - max_index
    - установить все значения по умолчанию.
    - (Сделать массив пустым, если min_index = max_index + 1).
 require
    valid_bounds: min_index <= max_index + 1
 ensure
    lower_set: lower = min_index
    upper_set: upper = max_index
    items_set: all_default
```

Как показывают первые два предложения в постусловии, процедура устанавливает *lower* и *upper* в соответствии с переданными в нашем примере значениями *your_lower_bound* и *your_upper_bound*. Они могут быть произвольными выражениями, например, константами:

```
create yearly_twentieth_century_revenue.make (1901,2000)
```

Здесь значения границ задаются в тексте программы, но они могут зависеть от переменных, входящих в выражения:

```
create another_array.make (m, m+n)
```

В приведенном примере интервал индексов имеет собственный смысл, задавая годы 20-го столетия. Если же просто необходимо создать массив из *n* значений, то нижняя граница обычно задается равной 1, а верхняя – *n*:

```
create simple_array.make (1,n)
```

Язык С и его последователи (С++, Java, C#) требуют, чтобы у всех массивов начальный индекс был равным 0. В примерах, таких как «годы 20-го столетия», это требует взаимных преобразований (сложение и вычитание 1901 в примере) между физическим индексом и его представлением. Для таких случаев, как *simple_array,* выбор между 0 и 1 – дело вкуса. Если вы, подобно мне, предпочитаете рассматривать большой палец на руке как первый, а не нулевой, а мизинец как пятый, а не четвертый, то выбор 1 кажется более разумным. Менее субъективный довод состоит в том, что начиная нумерацию элементов с нуля, приходится заканчивать ее номером *n*-1 для последнего элемента, и это, как показывает практика, является вечным источником ошибок.

Запрос *all_default* в последнем предложении постусловия выражает тот факт, что все элементы массива типа *ARRAY* [*SOME_TYPE*] будут после создания иметь значения по умолчанию, определяемые типом *SOME_TYPE*: ноль для *INTEGER* и *REAL*, *false* – для булевских, *void* – для любого ссылочного типа.

Доступ и модификация элементов массива

Приведем базисный запрос и команду для получения и модификации элемента массива:

```
item (i: INTEGER): G
        -- Элемент с индексом i, если это правильный индекс.
    require
        valid_key: valid_index (i)
put (v: like item; i: INTEGER)
        -- Изменить значение элемента с индексом i, если это правильный
            -- индекс, на значение v.
    require
        valid_key: valid_index (i)
    ensure
        inserted: item (i) = v
```

В обоих случаях предусловие требует, чтобы индекс находился в границах массива. Типичное применение команды, если уже объявлены переменные *your_array*: *ARRAY* [*SOME_TYPE*] и *your_value*: *SOME_TYPE*:

```
your_array.put (your_value, your_index)
```

Вызов метода *put* установил новое значение соответствующего элемента массива:

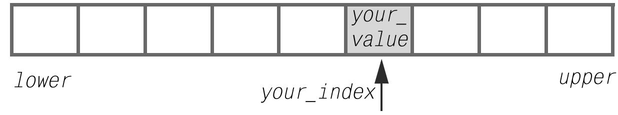


Рис. 13.5.

Заметьте порядок следования аргументов: сначала новое значение, затем индекс.
После этого вызова оператор

```
your_value:= your_array.item (your_index)
```

присвоит переменной *your_value* значение элемента массива с индексом *your_index*.

Постусловие *put* показывает, что непосредственно после выполнения *put* значение *item* с заданным индексом есть значение, заданное в *put*. Примеры с *put* и *item* корректны только при условии, что гарантируются «правильные» индексы. Если гарантии нет, то следует использовать вызовы в форме:

```
if your_array.valid_index (your_index) then
your_array.put (your_value, your_index)
else
…
end
```

Аналогично для *item*.

Для любой разумной реализации массивов вызов *put* и *item* выполняется за константное время – O(1). Это свойство используемой для хранения массивов RAM-памяти и причина широкого использования массивов.

Скобочная нотация и команды присваивания

Следующая нотация, применяющая квадратные скобки, доступна как для класса *ARRAY*, так и для некоторых других классов, изучаемых в этой главе:

```
your_value:= your_array [your_index]
    – Краткая запись для your_value:= your_array.item (your_index).
your_array [your_index]:= your_value
    – Краткая запись для your_array.put (your_value, your_index).
```

Это особенно удобно для таких операторов, как

```
a [i]:= a [i] + 1                                                  [3]
```

Скобочная запись значительно удобнее и следует математической традиции. Её преимущество особенно заметно в выражениях, включающих несколько элементов массива и операции над ними.

Ничего магического в скобочной записи нет, и она не является спецификой массивов. Для того, чтобы применить ее к любому типу, где это имеет смысл, достаточно включить сочетание **alias** "[]" после имени соответствующего метода при его объявлении. Именно это и сделано в классе *ARRAY* для *item*:

```
item(i: INTEGER) alias "[]": G assign put
        – Элемент с индексом i, если индекс правильный
    require
        valid_key: valid_index (i)
    do
        –… Реализация метода …
    end
```

Добавление **alias** “[]” к имени метода означает, что квадратные скобки являются псевдонимом имени метода (в данном случае – *item*) – еще одним способом вызова метода. В результате нотация

```
your_array [i]
```

это синоним (псевдоним) для

```
your_array.item (i)
```

В объявлении *item* также присутствует конструкция **assign** *put*. Любой запрос *q*, независимо от того, имеет ли он псевдоним в виде квадратных скобок, можно пометить **assign** *c*, где *c* – команда из того же класса, которому принадлежит запрос. Эффект состоит в том, чтобы сделать корректной нотацию, подобную присваиванию:

```
your_array.item(i):= your_value                                [4]
```

представляющую краткую запись вызова команды *put*

```
your_array.put (your_value, i)                                 [5]
```

Предложение **assign** связывает команду (*put*) с запросом (*item*). Такие команды называются *командами-присваивателями*.

> Терминологически команда, чья роль состоит в установке значения запроса, называется *сеттером*. *Сеттер* становится *присваивателем* благодаря языковому механизму, явно связывающему команду с запросом.

Поскольку *item* имеет скобочный псевдоним и связан с присваивателем, вполне законно использовать скобочную форму в последнем вызове:

```
your_array[i]:= your_value                                     [6]
```

Такая форма записи полностью согласуется с традиционной математической нотацией для массивов и векторов, используя в то же время семантику ОО-операций. Это сочетание и делает допустимым форму записи, использованную в [3].

Механизм команд-присваивателей применим к любым запросам, в том числе к атрибутам. Операторы **[4], [6]** хотя и имеют форму оператора присваивания, таковыми не являются, поскольку, как известно, скрытие информации запрещает прямое присваивание атрибутам (полям) класса. Они являются простыми вызовами процедур, эквивалентными **[5]** и соблюдающими все ОО-принципы. Это просто «синтаксический сахар», добавленный для удобства записи.

> Большинство языков программирования, начиная с Pascal, C и C++ до Java и C#, предлагают скобочную запись для массивов, как при доступе (*your_array* [*i*]), так и при модификации (*your_array* [*i*]:= *your_value*). В большинстве случаев эта форма является спецификой массивов, а сами массивы рассматриваются как встроенный в язык

специфический тип данных. В языке Eiffel *ARRAY* рассматривается как обычный класс с методами *item* и *put*, согласующийся с другими структурами данных и ОО-подходом (позволяя, например, наследование от класса *ARRAY*). Язык предлагает скобочную нотацию как псевдоним, благодаря конструкции **alias** «[]». Это общая конструкция, применимая не только к массивам. Она будет использоваться и при работе с другими структурами данных, в частности, с хэш-таблицами. Ее можно применять с тем же успехом при создании собственных классов.

Изменение размеров массива

В любой момент выполнения массивы имеют границы – *lower* и *upper*, следовательно, фиксированное число (*count*) элементов. Предусловие *valid_index* методов *put* и *item* отражает это свойство. В большинстве языков программирования это свойство массива устанавливается однажды и навсегда либо статически (используя константные границы), либо динамически в момент создания. В Eiffel можно перестраивать массив, используя метод *resize*:

```
resize (min_index, max_index: INTEGER)
            – Изменение границ массива, вниз к min_index
            – и вверх к max_index. Существующие элементы сохраняются.
      require
            good_indexes: min_index <= max_index
      ensure
            no_low_lost: lower = min_index.min (old lower)
            no_high_lost: upper = max_index.max (old upper)
```

Функции *min* и *max*, применяемые в постусловии, являются функциями над двумя целыми числами, дающими соответственно минимум и максимум из текущего числа, вызывавшего функцию, и ее аргумента.

Чаще всего для перестройки границ массива не вызывается специальный метод – она индуцируется при вызове метода *force*. Обычно, если нужно изменить значение элемента массива, базисным механизмом является метод *put*(*v*, *i*) с предусловием: *valid_index*(*i*). Это правильный способ работы, но при условии, что вы заранее знаете, какие элементы массива могут понадобиться. Если же в своих расчетах вы ошиблись, то это может привести к отказу в работе программы. В таких случаях для работы с массивом вместо *put* следует использовать метод *force*:

```
force (v: like item; i: INTEGER)
            – Заменить значение элемента массива на v, если индекс в допустимых
            – пределах.
            – Всегда применять перестройку границ массива, если индекс выходит
            – за пределы.
            –Сохранять существующие элементы
    ensure
            inserted: item (i) = v
            higher_count: count >= old count
```

В отличие от *put*, метод *force* не имеет предусловия, а потому всегда применим. Если *i* лежит вне интервала *lower*.. *upper*, процедура вызовет *resize* для изменения границ так, чтобы индекс оказался внутри нового интервала.

Из-за того, что реализация массива требует отведения ему непрерывного участка памяти, при перестройке массиву отводится обычно новый участок памяти и происходит копирование старых элементов массива:

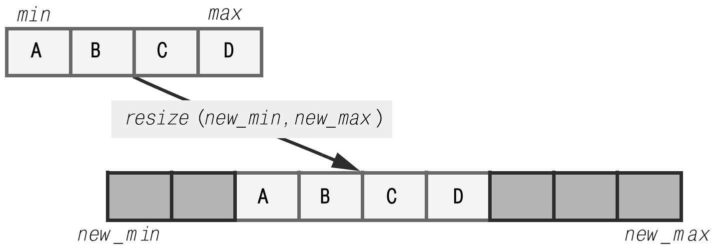


Рис. 13.6.

Перераспределение и копирование – это дорогие операции со сложностью O(*count*). В результате и *force* имеет сложность O(*count*), в то время как сложность *put* – O(1). Очевидно, что *force* следует использовать с осторожностью. Заметьте, что реализация *force* вполне разумна: она вызывает *resize* только при необходимости и, что более важно, изменяет размер массива с запасом, по умолчанию размер массива при перестройке увеличивается на 50%.

```
your_array.force (some_value, your_array.count + 1)
```

При повторных вызовах этого оператора *force* будет вызывать *resize* достаточно редко, выполняя в остальных случаях константное число операций.

Использование массивов

Массив типа *ARRAY* [*G*] представляет всюду определенную функцию, задающую отображение целочисленного интервала *lower..upper* в *G*. Если после создания границы *lower* и *upper* не изменяются или редко изменяются, то реализация высокоэффективна: так, доступ к значению и его модификация имеет сложность **O** (1) и, следовательно, работает быстро. Это делает массивы подходящими в тех случаях, когда:

- необходимо обрабатывать множество значений, ассоциированных с целочисленным интервалом;
- после создания массива и заполнения его начальными значениями доминирующими операциями будут индексно базируемые операции доступа и модификации.

Из-за высокой стоимости перераспределения массивы не подходят для высоко динамичных структур данных, где элементы приходят и уходят. В частности, операции вставки элемента в массив и удаление элемента являются дорогими операциями со сложностью O(*count*), поскольку это приводит к перенумерации элементов и их перемещению справа или слева от точки вставки или удаления. В таких ситуациях следует использовать другие структуры данных, которые будут рассмотрены позже в этой главе.

Производительность операций над массивами

Вот итоговая таблица, содержащая стоимость операций.

Операция	Метод в классе *ARRAY*	Сложность	Комментарий
Доступ по индексу	*item* **alias** “[]”	O(1)	
Замена по индексу	*put* **alias** “[]”	O(1)	
Замена по индексу вне текущих границ	*force*	O(*count*)	Требует перераспределения массива. Только небольшая часть последовательных выполнений метода будет причиной перераспределения
Вставка нового элемента		O(*count*)	Требует перенумерации индексов. У класса нет такого метода
Удаление элемента		O(*count*)	Требует перенумерации индексов. У класса нет такого метода

13.5. Кортежи

Массивы однородны: в экземпляре *ARRAY* [*T*] все элементы принадлежат типу *T* или типу, совместимому с *T*. **Кортежи (Tuples)** подобны массивам, но они могут содержать значения разных типов. Рассмотрим объявление:

```
tup: TUPLE [number: INTEGER, street: STRING, resident: PERSON]
```

Возможные значения *tup* во время выполнения представляют последовательности из трех компонентов, первый из которых имеет тип *INTEGER*, второй – *STRING,* третий – *PERSON*, предполагая, что такой класс существует. Такие кортежи полезны в различных приложениях, отражая тот факт, что в некотором доме с номером *number* на некоторой улице *street* живет гражданин *resident*.

Для задания значения кортежа достаточно записать в квадратных скобках последовательность значений соответствующих типов, разделяя элементы запятыми. Это значение можно использовать в качестве аргумента при вызове метода или присвоить переменной, такой как *tup*:

```
tup:= [99, "Rue de Rivoli", Louvre_museum_curator]                    [7]
```

Термин tuple в английский пришел из математики, где тройки, четверки называются *triple*, *quadruple* и по аналогии *n-tuple.* На русском мы говорим двойка, тройка и по аналогии *n*-ка (энка), но в переводах чаще используется более благозвучный термин «кортеж», понимаемый как последовательность из *n* значений.

Как типы кортежи не столь интересны, как массивы, списки, хэш-таблицы, бинарные деревья поиска, каждый из которых характеризуется своим собственным способом хранения и получения данных, своей эффективностью, преимуществами и ограничениями. Для кортежей наиболее распространенная реализация основана на массивах (игнорируя информацию о специфике типов компонентов кортежа, его можно рассматривать как *ARRAY*[*ANY*], где *ANY* — это общий универсальный тип высокого уровня, с которым совместимы все возможные типы). Поэтому для характеристики сложности операций над кортежами можно использовать уже известную нам таблицу для массивов.

Операция	Нотация	Сложность	Комментарий
Доступ к компоненту	*t.comp*	O(1)	Смотри ниже о нотации
Замена компонента	*t.comp:= value*	O(1)	
Вставка, удаление		Неприменима	

Интерес к кортежам в другом: этот механизм языка позволяет описать простым и ясным способом структуры данных без обращения к классам. В нашем примере **[7]**, где задавалось значение кортежа в целом, теги – *number*, *street*, *resident* – не играли роли, но они важны для доступа к индивидуальным значениям.

После выполнения **[7]** *tup.number* имеет значение 99. Теги можно также использовать и для задания значений компонентам, поскольку они рассматриваются как атрибуты с ассоциированными командами-присваивателями, что позволяет писать такие операторы, как:

```
tup.resident:= some_person
```

Конечно, можно было бы обойтись без типа *TUPLE*, используя классы, такие как:

```
class CENSUS_RECORD feature
        number: INTEGER assign set_number
        street: STRING assign set_street
        resident: PERSON assign set_resident
        set_number (n: INTEGER) do number:= n ensure number = n end
        … set_street, set_resident like set_number …
end
```

Над переменной *cr* типа *CENSUS_RECORD* были бы допустимы те же операции, что и над *tup*: доступ к полю (*cr.number*), модификация поля (*cr.resident:= some_person*).

Кортежи полезны тогда, когда все поля класса общедоступны, а сеттеры не имеют предусловий и только присваивают атрибутам значения, не делая ничего более. Такой частный случай класса описывает простую запись (составное значение, используемое, например, в реляционных базах данных). Использование кортежей в таких ситуациях избавляет нас от необходимости задавать классы, подобные *CENSUS_RECORD*. По этой причине кортежи называются ***анонимными*** классами. Если же с экземплярами кортежа необходимо выполнять более сложные операции, то следует переходить к настоящему классу.

Заметим, что теги не влияют на тип кортежа, фактически, они не обязательны. Ранее определенный кортеж можно задать как *TUPLE* [*a*: *INTEGER*, *b*: *STRING*, *x*: *PERSON*] или просто *TUPLE* [*INTEGER*,*STRING*, *PERSON*], если нет необходимости обращаться к компонентам по имени.

Синтаксически такие типы выглядят как универсально порожденные типы, подобные *LIST* [*T*]. Действительно, концепции очень похожи, но формально не существует класса *TUPLE,* поскольку это потребовало бы задать его с произвольным числом параметров, в то время как универсальный класс имеет фиксированное число параметров (один параметр у классов *ARRAY* [*G*] и *LIST* [*G*], два в *HASH_TABLE* [*G*, *KEY*]). Для кортежных типов можно описать последовательности любой длины: *TUPLE* без параметров задает все последовательности, *TUPLE* [*T*] – последовательность по меньшей мере из одного элемента, первый из которых имеет тип *T*, и так далее.

Это замечание определяет свойство *согласованности* кортежных типов: можно присваивать выражение типа *TUPLE* [*T*, *U*, *V*] переменной того же типа или любого из следующих типов – *TUPLE* [*T*, *U*]; *TUPLE* [*T*]; просто *TUPLE*. Последний из них (не класс, как отмечалось, но тип) покрывает задание всех возможных кортежей.

13.6. Списки

Список, известный также как *последовательность*, является контейнером, хранящим элементы в некотором порядке, обычно в порядке их вставки. Математически он представляет всюду определенную функцию, отображающую целочисленный интервал 1.. *count* в множество *G*. Это напоминает массивы, но разница в том, что *count* может свободно изменяться при вставке новых элементов.

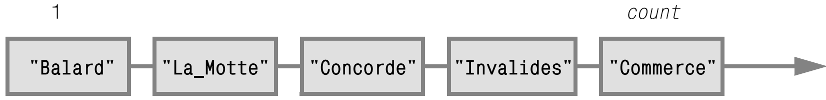


Рис. 13.7. Список

Рисунок показывает список, составленный из пяти элементов. Стрелка просто показывает, что порядок имеет значение. Те же элементы, но организованные в другом порядке, составляют другой список.

> Подобно массивам и другим структурам, где элементы упорядочены, мы систематически начинаем нумерацию с 1.

Возможны различные реализации списков. В EiffelBase они поддерживаются такими классами, как *LINKED_LIST*, *TWO_WAY_LIST*, *ARRAYED_LIST*, *MULTI_ARRAYED_LIST*. В данном разделе описываются свойства, общие для всех этих вариантов, заданные в классе *LIST*. Более подробно будет рассмотрен важный вариант связного списка, а затем будет дан обзор других вариантов. Реализация не будет обсуждаться во всех деталях, но примеры реализации и базисные идеи будут рассмотрены. Для получения полной картины следует обратиться к библиотеке классов EiffelBase и проанализировать соответствующие тексты.

> Класс *LIST* описывает общие абстрактные понятия, представляя пример отложенного класса, который прямо или косвенно наследуется другими классами. Это значит, что потомки не повторяют общие элементы, а получают их от класса более высокого уровня. Эти концепции будут подробно обсуждаться в главе, посвященной наследованию.

Классы, задающие списки, рассматривают список не просто как коллекцию элементов, но как *машину*, которая в любой точке своего существования имеет состояние, характеризуемое *курсором*:

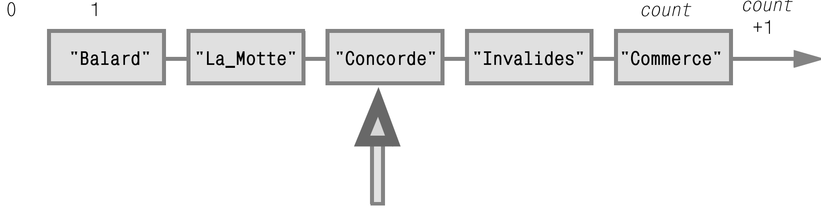


Рис. 13.8. Список с курсором

Это не новое понятие. Когда мы рассматривали линии метро как список станций, мы уже встречались с курсором.

Наличие курсора облегчает выполнение базисных операций – доступ, вставку, удаление элементов, позволяя соответствующим методам иметь более простой интерфейс – действия будут выполняться над элементом, заданным позицией курсора, так что им не требуется дополнительное указание позиции элемента. Конечно, это требует, чтобы существовала возможность перемещения курсора к нужному элементу.

В этой схеме курсор является *внутренним* понятием – у каждого списка свой собственный курсор. Возможно введение объектов, задающих внешний курсор; таких курсоров может быть несколько, позволяя различным клиентам списка рассматривать собственное представление списка. В частности, это полезно в параллельном программировании. Для деталей следует обратиться к библиотеке EiffelBase, рассмотрев класс *CURSOR* и его потомков. Типичное применение внешнего курсора иллюстрируется в последующих главах, где внешний курсор используется для запоминания начального состояния внутреннего курсора и его восстановления после завершения операции, требующей проход по списку.

Запросы, связанные с курсором

Мы должны позволять курсору списка, показанному на последнем рисунке, занимать позиции из расширенного интервала 0.. *count* + 1, а не из интервала 1.. *count*, где находятся элементы списка. Курсор может находиться левее первого элемента и правее последнего элемента списка. В полезности такого представления легко убедиться. Позицию курсора дает запрос:

```
index: INTEGER
        – Текущая позиция курсора.
```

Предложения, задающие инвариант, выглядят так:

```
non_negative_index: index >= 0
index_small_enough: index <= count + 1
```

Для характеристики двух экстремальных случаев позиции курсора введены два запроса:

```
before: BOOLEAN
          – Правда, что слева от курсора нет правильной позиции?
after: BOOLEAN
          – Правда, что справа от курсора нет правильной позиции?
```

Обратите внимание на корректность формулировок комментариев, которые должны соответствовать всем возможным случаям, в частности, учитывать возможность пустого списка.

В соответствии со стандартом стиля, принятым для булевских запросов, наши запросы должны именоваться как *is_before*, *is_after*, но они уже так давно используются, что переименовывать их «рука не поднялась». Другие запросы, примененные ниже, такие как *is_empty*, следуют нормальному соглашению.

Если в текущем состоянии курсор списка находится в позиции *before* или *after*, то мы говорим, что он «off»:

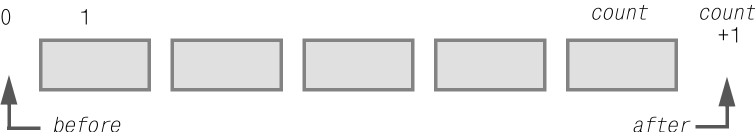


Рис. 13.9. Позиция курсора before и after

```
off: BOOLEAN
        — Верно ли, что курсор вне списка?
    ensure
        definition: Result = (after or before)
```

Следующие предложения инварианта выражают свойства этих запросов (возможны также и соответствующие постусловия):

```
before_definition: before = (index = 0)
after_definition: after = (index = count + 1)
off_definition: off = (index = 0 or index = count + 1)
```

Другие запросы о курсоре:

```
is_first: BOOLEAN
            — Задает ли курсор первый элемент?
    ensure
                valid_position: Result implies (not is_empty)
is_last: BOOLEAN
            — Задает ли курсор последний элемент?
    ensure
            valid_position: Result implies (not is_empty)
```

Еще один важный запрос:

```
is_empty
        — Пуст ли список?)
```

Список может пустым, и тогда запросы *is_first* и *is_last* будут ложными; курсор может быть на первом (последнем) элементе, если в списке есть хотя бы один элемент. Следует всегда помнить о принципе экстремальных случаев, требующем обращать особое внимание на граничные ситуации. В отсутствии элементов рисунок, иллюстрирующий список, выглядит так:

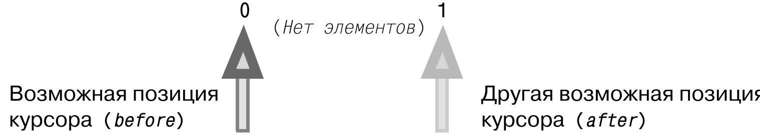


Рис. 13.10. Пустой список с его двумя возможными позициями курсора

В этом случае *count* равно нулю и максимальная позиция *index*, удовлетворяющая инварианту, *count* + 1, равна 1. В таком пустом списке курсор может быть либо в позиции 0, либо в позиции 1. В любом случае *off* будет иметь место, что следует из предложения инварианта:

```
empty_constraint: is_empty implies off
```

Заметьте повторяющееся накопление предложений инварианта, позволяющее выразить шаг за шагом наше понимание используемых структур объектов.

Почувствуй методологию: использование инвариантов

Используйте предложения инварианта, чтобы сделать явными свойства проектируемых классов. Проверяйте логичность и совместимость свойств друг с другом (в частности, рассматривайте граничные ситуации в соответствии с принципом экстремальных случаев).

Для получения доступа к элементу в позиции курсора

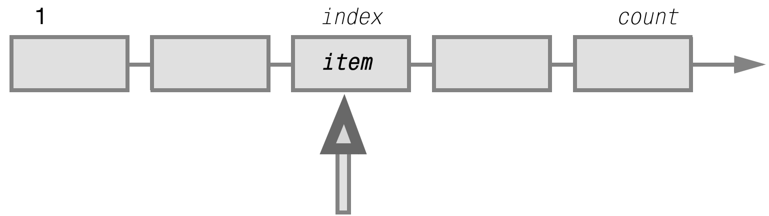


Рис. 13.11. Текущий элемент

используйте запрос

```
item: G
        -- Элемент в позиции курсора.
    require
        not_off: not off
```

Этот запрос возвращает результат типа *G*, родовой параметр классов, задающих списки (*LIST* [*G*], *LINKED_LIST* [*G*] и т.д.). Обратите внимание на предусловие: в состоянии *off* (пустой список) нет текущего элемента.

Перемещение курсора

В нашем распоряжении есть несколько команд, позволяющих перемещать курсор. Вот команды, позволяющие устанавливать курсор в начало или в конец списка:

```
start
            -- Переместить курсор в первую позицию
            -- (не выполняется для пустого списка)
    ensure
            at_first: (not is_empty) implies is_first
finish
```

```
            — Переместить курсор в последнюю позицию
            — (не выполняется для пустого списка)
      ensure
            at_last: (not is_empty) implies is_last
```

Вызов *start* означает истинность *is-first*, а вызов *finish* означает *истинность is-last*. Для пустых списков это не справедливо, что отражено в постусловии. Курсор можно также перемещать на одну позицию вправо и влево:

```
forth
            — Переместить курсор в следующую позицию.
      require
            not_after: not after
      ensure
            moved_forth: index = old index + 1
back
         — Переместить курсор в предыдущую позицию.
      require
            not_before: not before
      ensure
            moved_back: index = old index - 1
```

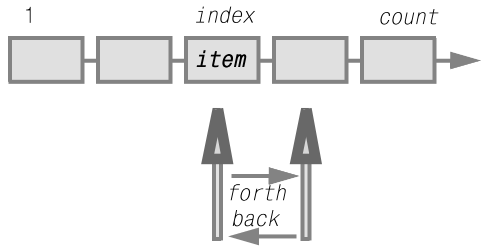


Рис. 13.12.

Предусловия гарантируют, что индекс остается в границах, заданных предыдущими предложениями инварианта: *non_negative_index* и *index_small_enough*. Курсор можно установить в заданную позицию:

```
go_i_th (i: INTEGER)
            — Переместить курсор в i-ю позицию.
      require
            valid_cursor_position: i >= 0 and i <= count + 1
      ensure
            position_expected: index = i
```

Итерирование списка

Часто приходится применять одну и ту же операцию ко всем элементам списка. Предположим, что операция задается методом:

```
your_operation (x: G)
```

С этой ситуацией мы уже встречались при рассмотрении линий метро – общую форму задает цикл:

```
from
    your_list.start
until
    your_list.after
loop
    your_operation (your_list.item)
    your_list.forth
variant
    your_list.count - your_list.index + 1
end
```

Здесь операция применяется к некоторому существующему в вашей программе списку *your_list*. Эта же схема будет использована и в классах, задающих списки, для прохода по *текущему* списку, но в этом случае вызовы будут неквалифицированными – просто *start*, *after*, *forth* без *your_list*. Примеры скоро появятся в методах *search* и has, где разыскиваются нужные элементы списка.

Существуют вариации этой схемы, например, операция применятся к элементам списка, пока не встретится элемент, удовлетворяющий некоторому условию:

```
from
    your_list.start
until
    your_list.after or else your_condition (your_list.item)
loop
    your_operation (your_list.item)
    your_list.forth
variant
    your_list.count - your_list.index + 1
end
```

Такие схемы являются примером итерирования структуры данных.

Определение: итерирование

Итерированием структуры объектов является применение заданной операции ко всем элементам структуры или ко всем элементам, удовлетворяющим заданному условию.

Другой термин для «итерирования» – это «*обход*». *Итерация* – это применение механизма итерирования к структуре, хотя часто этот термин используется, когда речь идет об одном шаге процесса («на каждой итерации цикла курсор передвигается на одну позицию»). Итератор – это механизм, преобразующий операцию над отдельным элементом в операцию над всеми элементами структуры.

Примером реализации, использующей механизм итераций, который разделяется всеми классами, задающими списки, является процедура *search*, осуществляющая поиск элемента в списке. Ее текст выглядит так:

```
search (v: G)
        —Если курсор установлен в позиции before и список не пуст,
        — то курсор передвигается в начало списка.
        — При поиске курсор устанавливается на первом элементе, совпадающем с v.
        — Если такового  нет, то курсор переходит в позицию after.
        do
            from
                if before and not is_empty then
                        forth
                end
            until
                    after or else item = v
            loop
                forth
            end
        end
```

Этот метод является командой, перемещающей курсор:

- если заданное значение *v* встречается в текущей позиции или в позиции справа от курсора — то к первой такой позиции;
- в противном случае — к граничной позиции справа (*after*).

Это соглашение позволяет использовать метод *search* повторно для поиска последовательных вхождений значения. Процедура также применяется в реализации запроса *has*, отвечающего на вопрос о существовании в списке элемента с заданным значением:

```
has (v: G)
                — Содержит ли структура вхождение v?
        local
            original_index: INTEGER
        do
            original_index:= index
            start
            search (v)
            Result:= not after
            go_i_th (original_index)
        end
```

Будучи запросом, *has* должна оставлять структуру в состоянии, предшествующем запросу. Поэтому здесь вводится локальная переменная *original_index*, запоминающая начальную позицию курсора и восстанавливающую ее в конце работы, благодаря методу *go_i_th*.

Как *search*, так и *has* требуют O(*count*) времени как в среднем, так и в максимуме.

Итерационная схема, иллюстрируемая *search*, многократно встречается при работе со списками и другими последовательными структурами; мы уже встречались с ней в нескольких примерах, начиная с цикла, подсчитывающего общее время проезда по линии метро.

В конце этой главы мы вернемся к концепциям итерации и получим первое представление об общих механизмах, позволяющих применять стандартный механизм итерирования вместо того, чтобы каждый раз задавать явный цикл с *start*, *forth*, *item* и *after*.

Добавление и удаление элементов

Для добавления элемента в список – в его начало, конец или в позицию курсора – можно использовать одну из операций со следующими спецификациями:

```
put_front (v: G)
            – Добавить v в начало, не перемещая курсор.
put_left (v: G)
            – Добавить v слева от позиции курсора, не перемещая курсор.
      require
            not_before: not before
put_right (v: G)
            – Добавить v справа от позиции курсора, не перемещая курсор.
      require
            not_after: not after
extend (v: G)
            – Добавить v в конец, не перемещая курсор.
```

Как показывают комментарии, эти процедуры спроектированы так, чтобы не оказывать воздействия на курсор, поскольку нет причин, чтобы вставка изменяла текущую активную позицию курсора.

Во многих случаях реализация должна временно изменять позицию курсора, например, можно следующим образом реализовать расширение списка *extend*(*v*):

```
original_index:= index
finish
put_right (v)
go_i_th (original_index)
```

Здесь, как и в *has*, записывается начальная позиция, которая восстанавливается в конце.

Для удаления элементов можно использовать:

```
remove
         – Удалить элемент в позиции курсора; передвинуть курсор к правому
         – соседу (или к after, если правого соседа нет).
      require
         item_exists: not off
      ensure
         removed: count = old count - 1
         after_when_empty: is_empty implies after
```

В этом случае курсор должен быть передвинут, поскольку исчез элемент, на который указывал курсор.

Разрешается удалять элементы слева или справа от курсора, не изменяя при этом позицию курсора. В качестве упражнения напишите реализацию методов *remove_left*, *remove_right*, задав их спецификацию (сигнатуру, заголовочный комментарий, контракт).

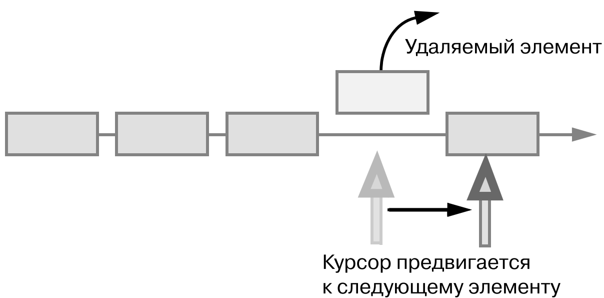


Рис. 13.13. Удаление текущего элемента

13.7. Связные списки

Мы уже познакомились с базисными свойствами и методами списков, независимо от того, как они реализованы. Займемся теперь вариантами, начав с важного частного случая связного списка, с класса *LINKED_LIST*.

Основы связных списков

При работе со станциями метро мы познакомились с приемами связывания элементов в последовательную структуру:

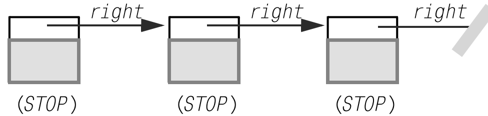


Рис. 13.14. Связывание станций

Теперь, благодаря механизму универсализации, возможно обобщение на произвольные структуры. Экземпляр *LINKED_LIST*[*T*] для некоторого типа *T* будет ссылаться на ноль или более связанных ячеек, принадлежащих классу *LINKABLE*[*T*]. Каждый экземпляр класса *LINKABLE*[*T*] содержит значение типа T и ссылку на другой экземпляр этого класса:

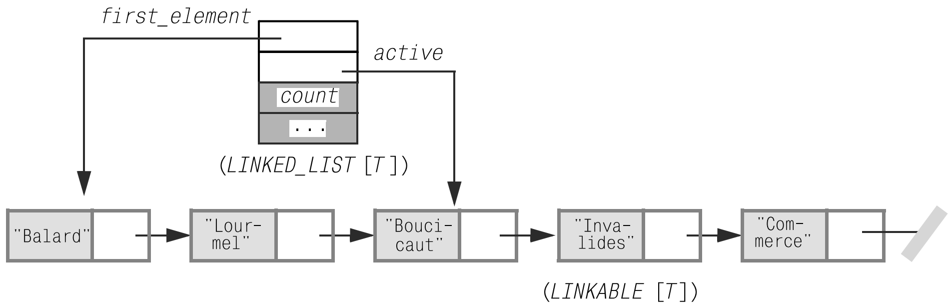


Рис. 13.15. Связный список

Как показано на рисунке, реализация включает два класса.

- Верхний объект является экземпляром *LINKED_LIST*[*T*]. Такой объект называют заголовком списка – он содержит общую информацию о списке и обеспечивает доступ к элементам списка, но сам он не представляет никакого элемента. Поле *count* задает число элементов. Оно реализовано атрибутом (но могло быть также и функцией). Другие поля являются ссылками на ячейки списка. Атрибут *first_element* задает ссылку, ведущую к первой ячейке, *active* – ведет к ячейке в позиции курсора.
- Другие объекты представляют ячейки списка; они являются экземплярами класса *LINKABLE*, также универсального, с тем же родовым параметром, задающим тип элементов списка – *LINKABLE*[*T*].

При нормальном использовании клиентскому приложению требуется только класс *LINKED_LIST*. Класс *LINKABLE* необходим для реализации, представляя очень простое понятие ячейки списка, которая может быть связана с другой подобной ячейкой: типичный экземпляр выглядит следующим образом:

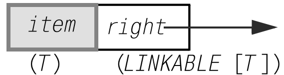


Рис. 13.16. Экземпляр LINKABLE[T].

Реализация методов класса *LINKED_LIST* основана на запросах класса *LINKABLE* и связанных сеттер-командах:

```
item: G
        – Значение в ячейке.
right: LINKABLE [G ]
        – Следующий элемент.
put (x): G
            – Установить значение элемента равным x.
    ensure
            set: item = x
put_right (other: LINKABLE [G])
            – Связывание с ячейкой other.
    ensure
            set: right = other
```

Вставка и удаление

Следующий рисунок показывает, как класс *LINKED_LIST* реализует команду *put_right*, которая должна добавлять элемент справа от курсора, не перемещая сам курсор. Для связного списка достаточно создать новую ячейку *LINKABLE* и обновить ссылки:

Реализация метода использует два вызова *put_right* из класса *LINKABLE*. Обратите внимание также на то, что требуется особый разбор случая, когда список пуст и *before* истинно:

```
put_right (v: G)
        – Добавить v справа от позиции курсора, не меняя его положения.
    require
        not_after: not after
```

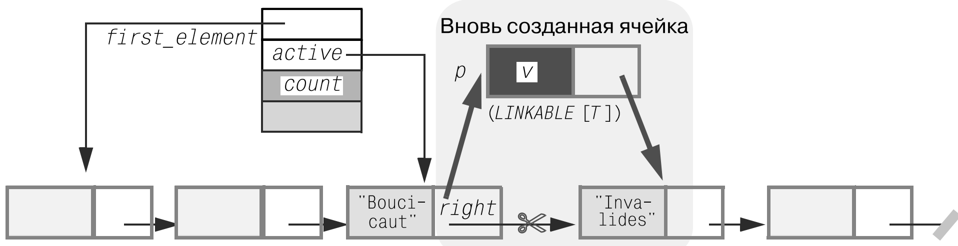


Рис. 13.17. Добавление ячейки

```
        local
            p: LINKABLE [G ]        — Ячейка должна быть создана
        do
            create p.make (v)
            if before then             — Специальный случай before:
                p.put_right ( first_element)
                first_element:= p
                active:= p
            else                          — Общий случай:
                p.put_right (active.right)
                active.put_right ( p)
            end
ensure
            next_exists: active.right /= Void
            inserted: (not old before) implies active.right.item = v
            inserted_before: (old before) implies active.item = v
        end
```

В данном примере мы манипулируем со ссылками. Он еще раз демонстрирует, как необходимо проявлять аккуратность при проведении этих операций. Даже здесь, когда выполняются простые операции, алгоритм требует особого внимания, чтобы убедиться, что он корректно работает во всех ситуациях. Трудности возрастают в следующих двух примерах – удаления элемента и обращения списка.

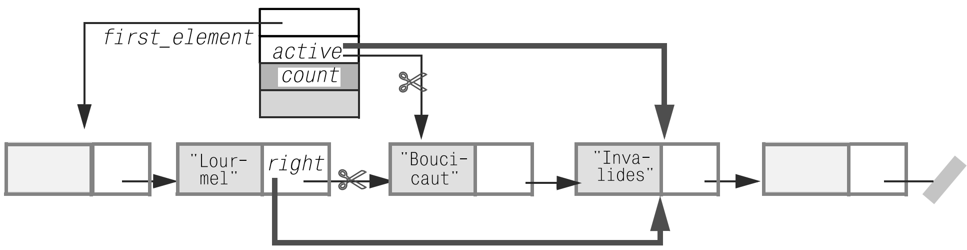


Рис. 13.18. Удаление ячейки

Процедура *remove* удаляет ячейку списка в позиции курсора. Спецификация, заданная ранее, устанавливает, что курсор передвигается в следующую позицию, непосредственно справа от той, где он находился, что показана на следующем далее рисунке. Реализация должна изменить две ссылки.

- Необходимо изменить ссылку *right* у ячейки, непосредственно предшествующей курсору (у элемента со значением «Lourmel»), чтобы обойти удаляемый элемент.
- Обновить в соответствии с требованиями позицию курсора, для чего следует изменить ссылку *active* объекта *LINKED_LIST* (здесь элемент «Invalides»).

Для изучения текста процедуры следует обратиться к фактической реализации – процедуре *remove* из класса *LINKED_LIST*. Она более сложная, чем *put_right*, поскольку необходимо учитывать больше специальных случаев, в частности, когда курсор установлен на первом или последнем элементе. Целесообразно вначале разобрать текст более простой процедуры – *remove_right*.

Обращение связного списка

В качестве заключительной иллюстрации алгоритмов, манипулирующих ссылками, рассмотрим метод, единственный в этой главе, не включенный в момент написания этого текста в состав соответствующего класса *LINKED_LIST* библиотеки EiffelBase (нет никаких объективных причин, чтобы не сделать это теперь). Мы хотим создать *процедуру*, обращающую элементы списка. Основная идея ясна, поскольку мы уже писали *функцию*, решающую эту задачу. Теперь мы должны справиться с двумя дополнительными проблемами: обобщением на произвольные связные списки и необходимостью обращения существующего списка на том же месте, что делает нашу задачу более сложной.

Прежде всего заметим, как *не* следует решать задачу. Корректный, но неэффективный алгоритм может быть следующим. Вначале нужно перейти к последнему элементу, связать с ним *first_element*-ссылку, а сам элемент связать с его левым соседом. Затем повторять эти действия, проходя каждый раз с левого конца по ссылкам до достижения обращенной части списка. При этом понадобятся две локальные переменные, одна будет хранить старое значение *first_element*, чтобы можно было выполнить проход по списку, вторая будет хранить ссылку на левого соседа, необходимую для обращения текущего элемента.

Время программирования!

Алгоритм обращения со сложностью $O(n^2)$

Превратите приведенное неформальное описание в процедуру и испытайте ее на некотором списке целых. Организуйте код так, чтобы подсчитывалось число итераций цикла.

Было бы неправильно модифицировать библиотечный класс *LINKED_LIST*, так что нужно поступить правильно и создать наследника – собственный небольшой класс, где можно выполнять желаемые модификации:

```
class MY_INTEGER_LIST inherit LINKED_LIST [INTEGER] feature
            reverse do … Разместите здесь Ваш код … end
end
```

Нет необходимости изменять другие свойства родительского класса.

Проблема такого подхода связана с его эффективностью: первый проход по списку потребует *count* итераций, второй *count* – 1, третий *count* – 2 и т.д., так что общее число итераций равно *count* (*count* + 1) / 2, что означает **O** (*count*^{2}). Мы же хотим иметь алгоритм со сложностью **O**(*count*). Начнем его проектировать. Он начинается так же, как и версия для функции с единственным циклом, но изменяет структуру цикла на месте, не создавая нового списка:

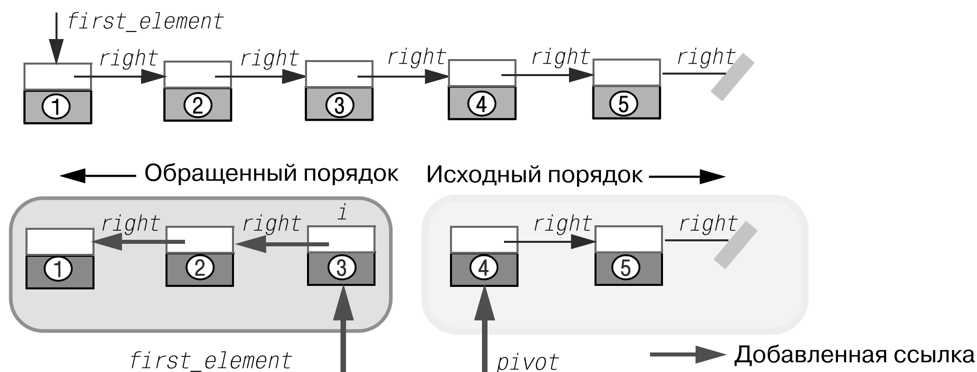


Рис. 13.19. Обращение связного списка: промежуточное состояние (исходное состояние показано вверху)

В этом промежуточном состоянии:

- *first_element* присоединен к одному из элементов исходного списка (как на рисунке) или имеет значение void;
- мы полагаем, что *first_element* всегда представляет позицию *i* в исходном списке: позицию элемента или, если *first_element* является void, позицию слева от первого элемента при условии его существования (это также означает, что для пустого списка – помните о проверке граничных условий! – *first_element* является void);
- *pivot* присоединен к элементу, который был непосредственным правым соседом элемента *i* в исходном списке. Согласно правилам, *pivot* есть void, если и только если либо *i* был последним элементом исходного списка, либо список пуст;
- начиная с *pivot* и следуя *right*-ссылкам, представлен список, который составлен из всех элементов исходного списка, следующих за элементом *i*, если они есть в их **исходном** порядке;
- начиная с *first_element* и следуя *right*, представлен список, который составлен из всех элементов исходного списка, предшествующих, а также включая элемент *i*, если таковые есть в их **обращенном** порядке.

Здесь присутствуют все признаки хорошего инварианта итеративного процесса, где процесс может рассматриваться как последовательная аппроксимация. Достаточно тривиально обеспечить начальную истинность инварианта, установив *pivot* как исходный *first_element* и *first_element* – void. Инвариант вырабатывает желаемый результат по завершении, когда *i* является последней позицией исходного списка, *first_element* даст нам полностью обращенный список! Когда же инвариант выполняется в промежуточном состоянии, то несложно расширить его на следующий элемент списка, изменяя значение трех ссылок, как показано на следующем рисунке:

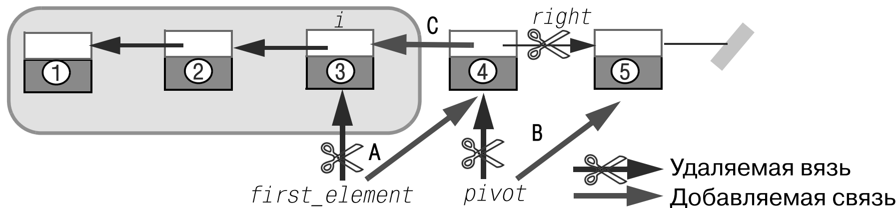


Рис. 13.20. Обращение связного списка: добавление одного элемента

Код такой итерации цикла достаточно прост:

```
i:=first_element
first_element:= pivot            – Операция, помеченная A на рисунке
pivot:= pivot.right          – Операция B
first_element.put_right (i)          – Операция C
```

Заметьте необходимость временной переменной *i* для записи исходного значения *first_element*, так, чтобы мы могли связать новый первый элемент в последней операции C.

Приведем полный алгоритм, соединяя все вместе. Для выражения варианта – следует заботиться о завершаемости цикла – добавим целочисленную переменную *c*, подсчитывающую итерации, или, что то же, записывающую позицию текущего *first_element*.

```
reverse
          – Изменить связи элементов в обратном порядке.
          – (Нет предусловия - будет работать и для пустого списка.)
          – Не перемещает курсор.
      local
          pivot, i: LINKABLE [G ] ; c: INTEGER
      do
          from
             pivot:= first_element ; first_element:= Void ; c:= 0
          invariant
          – c - индекс элемента first_element, если он есть в исходном списке;
             – список, начинающийся с first_element, включает все элементы
             – в обращенном порядке вплоть до позиции c в исходном списке;
             – список, начинающийся с pivot, включает все элементы в исходном
             –порядке, начиная с позиции c.
          until
             pivot = Void
          loop
             i:= first_element
             first_element:= pivot
             pivot:= pivot.right
             first_element.put_right (i)
             c:= c + 1
          variant
             count - c + 1
```

```
    end
end
```

Убедитесь, что вы хорошо понимаете алгоритмы *reverse* и *put_right*, поскольку они демонстрируют основные идеи реализации операций над связными списками. Они также демонстрируют трудности, возникающие при работе со ссылками, и напоминают о принципе программирования ссылок – такие операции должны быть частью специальных, тщательно спроектированных кластеров или частью профессиональных библиотек общецелевого назначения, но не должны быть частью приложения, реализующего «бизнес-логику» программы.

Для тестирования понимания алгоритма обращения списка следует написать его варианты для реализаций, изучаемых далее, – списка, построенного на массивах, и двусвязного списка.

Производительность операций над связным списком

Можно оценить стоимость операций над связным списком.

- Операции, для которых нужно выполнять действия в позиции курсора, – *put_right* и *remove_right* – имеют сложность O(1).
- Операции, которым необходим проход по списку, имеют сложность O(*count*). К таким операциям относятся независимо от реализации *search* и *has*. Процедура *reverse*, как мы видели, также имеет сложность O(*count*). Такую же сложность имеет и операция по перемещению курсора – *go_i_th,* а также *finish*, реализованная как *go_i_th* (*count*).

Интересный случай представляет операция *extend*, добавляющая элемент в конец списка. Как отмечалось, она может быть реализована через операцию *finish*, за которой следует операция *put_right* и *go_i_th*, если требуется восстановить позицию курсора. Поэтому суммарная сложность операции есть O(*count*). Часто при создании списка приходится поочередно добавлять новые элементы в конец списка. В этом случае позволительно оставлять курсор в конце списка, тогда для реализации *extend* нужно выполнить операции *put_right* и *forth*, что дает сложность O(1).

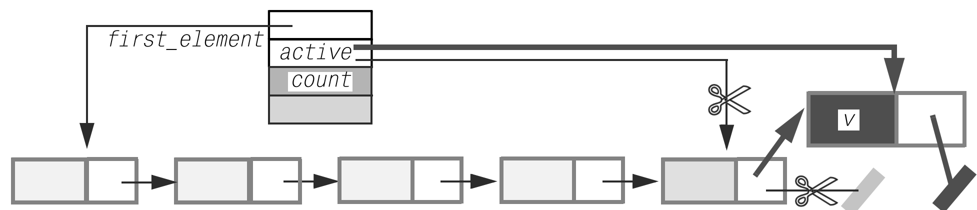


Рис. 13.21. Вставка в конец

Некоторые операции, работающие в позиции курсора, более хитро устроены, чем *put_right* и *remove_right*: им может, например, понадобиться ссылка на элемент, стоящий слева от курсора. Таковой является операция *remove*, удаляющая элемент под курсором. Включение атрибута *previous*, указывающего на левого соседа, позволяет сохранить сложность O(1) для таких операций, но несколько снижает эффективность других операций, поскольку требует обновления атрибута при выполнении ряда действий.

Интерфейс *LINKED_LIST* и других списковых классов не делает очевидного различия между *forth*, передвигающим курсор на одну позицию вправо, и *back*, двигающим курсор в

обратном направлении. Однако производительность этих операций кардинально отличается: *forth* – O(1), в то время как *back* имеет сложность O(*count*), будучи реализована как

```
start
go_i_th (index - 1)
```

При добавленном атрибуте *previous* сохраняется сложность O(1), но только при однократном выполнении *back*, так что в общем случае введение этого атрибута мало чем помогает в эффективности этой операции. Симметричные структуры, такие как двусвязные списки *TWO_WAY_LIST*, рассматриваемые ниже, устраняют эти трудности.

Дадим обзор сложности выполняемых операций. Вначале рассмотрим операции вставки удаления.

Операции	**Методы в классе** *LINKED_LIST*	**Сложность**	**Комментарий**
Вставка в позицию курсора	*put_right, put_left*	O(1)	Для операции слева от курсора сложность O(1) требует атрибута *previous*
Удаление в позиции курсора	*remove_right, remove*	O(1)	*remove_left* имеет сложность O(*count*)
Вставка в конец, если курсор находится уже там	*extend*	O(1)	

Сложность операций по изменению позиции курсора:

Передвинуть курсор к первому элементу	*start*	O(1)
Передвинуть курсор к последнему элементу	*finish*	O(*count*)
Передвинуть курсор на шаг вправо	*forth*	O(1)
Передвинуть курсор на шаг влево	*back*	O(*count*)

Глобальные операции могут требовать прохода по списку:

Вставка в конец, если курсора там нет	*extend*	O(*count*)	
Поиск	*search, has*	O(*count*)	
Обращение	*reverse*	O(*count*)	Нет в классе, но описана выше

13.8. Другие варианты списков

Класс *LINKED_LIST* представляет одну из возможных реализаций списков. Существуют и другие варианты.

Двусвязный список

В реализации *LINKED_LIST* предпочтение отдается проходу по списку слева направо, и как следствие, возникает существенное различие в эффективности методов *forth* и *back*. Класс *TWO_WAY_LIST* представляет полностью симметричную структуру. Платой за это являются потери в памяти, так как вместо *LINKABLE* с одной связью приходится использовать класс *BI_LINKABLE,* в котором каждая ячейка имеет две ссылки – вперед и назад:

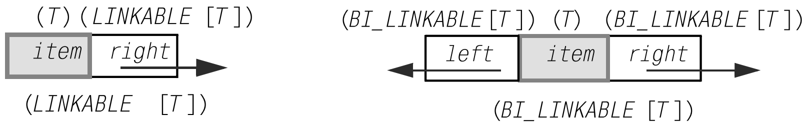


Рис. 13.22. Экземпляры LINKABLE [T] и BI_ LINKABLE [T]

Класс *TWO_WAY_LIST* содержит не только ссылку *first_element*, но и, дополняя симметрию, ссылку *last_element* на последний элемент списка:

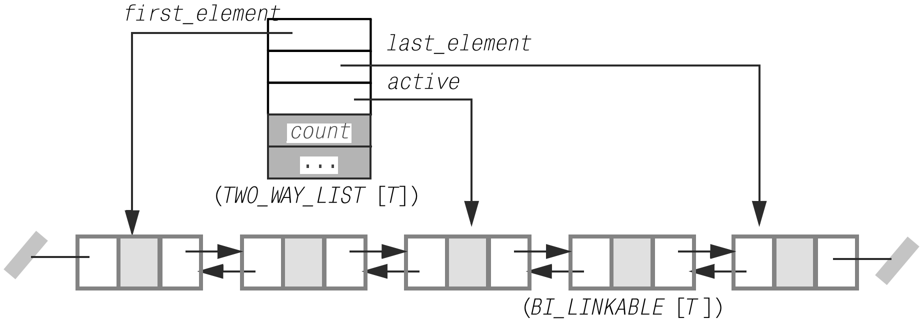


Рис. 13.23. Двусвязный список

В результате симметрии операции, требующие левого просмотра, такие как *remove_left* и *put_left*, так же как и их правосторонние двойники, имеют эффективность O(1), а метод back становится так же хорош, как и метод *forth*.

Абстракция и ее следствия

Здесь я должен рассказать небольшую историю с полей сражения. Как-то программисты одной из компаний объясняли своему менеджеру, что объектно-ориентированный инструментарий слишком медленный. Менеджер попросил старшего разработчика провести инспекцию кода, в результате чего было обнаружено, что на экземплярах *LINKED_LIST* многократно выполняется операция *back*. Заменив этот класс на *TWO_WAY_LIST*, получили ускорение работы в 23 раза. После этого программисты жили счастливо и никогда не говорили о потере скорости генерируемого кода.

Мораль: *абстракция* – краеугольный камень современного программирования – учит двум важным урокам.

Во-первых, она показывает существование рисков: вполне разумно и красиво рассматривать *back* как операцию, применимую ко всем спискам. Действительно, мы видели, что *back*

реализована для односвязных списков. Если рассматривать список просто как список, абстракция с ее техникой полиморфизма и динамического связывания позволяет забыть, что иногда мы имеем дело с двусвязным списком, где *back* имеет эффективность O(1), а иногда может появляться односвязный список, возвращающий *back* в медлительное сообщество методов O(*n*). Наш первый урок состоит в том, что в практике профессиональной разработки ПО, где производительность является одним из важнейших ограничений, не следует позволять преимуществам функциональной абстракции подавлять свойства эффективности.

Если первый урок высвечивает возможную темную сторону абстракции, то второй урок свидетельствует о ее заслугах. Для достижения ускорения достаточно было изменить всего лишь несколько объявлений, заменив тип *LINKED_LIST* на *TWO_WAY_LIST*. Без ОО-методов и абстракции детали реализации были бы глубоко запрятаны внутрь приложения и изменения были бы более существенными и трудными.

Абстракция не враг производительности. Она может изначально спрятать проблемы производительности от невнимательного наблюдателя, но она вместе с тем позволяет эффективно обнаружить эти проблемы и скорректировать любые несоответствия, которые вы обнаружили.

Списки на массивах

Для представления списков можно использовать массивы вместо ссылочных структур:

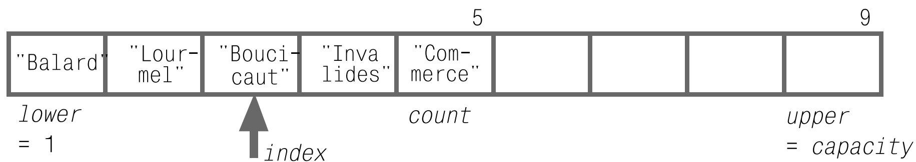


Рис. 13.24. Массив, реализующий список

Класс *ARRAYED_LIST* библиотеки EiffelBase обеспечивает эту реализацию. Пусть вас не смущает, что используется класс *ARRAY*: экспортируемые методы, видимые клиентам *ARRAYED_LIST*, те же, что и у класса *LIST*, реализуются методами класса *ARRAY*, такими как *item* и *put*. Внутренне, как показано на рисунке, *lower* равно 1, так что имеет место следующий инвариант класса: *capacity* = *upper* – *lower* + 1, откуда следует, что *capacity* = *upper,* так что верхняя граница задает физический размер массива.

У массива число его элементов *count* совпадает с его емкостью *capacity.* Но это не верно для *списка, построенного на массиве*: *count* является числом элементов списка, в то время как *capacity* – это максимально возможное число элементов, которое может изменяться при необходимости. На приведенном выше рисунке *count* равно 5, а *capacity* равно 9, и индексы элементов списка находятся в пределах от 1 до *count.* Инвариант класса включает свойство *count* <= *capacity*.

Курсор задается целочисленной переменной *index*, которая в классе *ARRAYED_LIST* реализована как атрибут. Еще один признак, отличающий массив от списка, построенного на массиве, состоит в том, что индекс массива меняется от 1 до *capacity* (от *lower* до *capacity*), но *index* может меняться, как принято в списках, от 0 до *capacity* + 1.

Реализация не обязана размещать элементы списка, начиная с нижней границы. Для поддержки левосторонней вставки полезно рассматривать массив как круговую структуру с двумя маркерами на концах, – эта техника будет описана при рассмотрении круговых очередей. Реализация не представляет сложностей, однако позволяет для очередей избавиться от фа-

тального ограничения списков на массивах – вставка и удаление требуют перемещения всех элементов слева или справа от курсора. Эти операции дорогостоящие и требуют O(n) времени. Удаление элементов можно откладывать, оставляя пустоты, но в какой-то момент потребуется провести O(n) сжатие массива. Если при вставке окажется, что достигнута емкость массива, то придется массив перестраивать, увеличивая емкость, а это, как отмечалось ранее, дорогостоящая операция.

Эти свойства существенно ограничивают полезность списков, построенных на массивах. Они представляют интерес для определенного круга сценариев, где изначально создается список, после чего он остается в относительно стабильном положении с небольшим числом вставок и удалений. Тогда список на массивах дает преимущества, особенно если часто выполняются операции по произвольному доступу к элементам по индексу – операция *go_i_th*(i) имеет сложность O(1), а не O(n), как для связных списков.

Мультимассивные списки

На тему списков вариаций может быть множество. Одной из таких вариаций является класс *MULTI_ARRAY_LIST*, соединяющий преимущества массивов и связных списков. Платой за это служит дополнительная сложность, как можно видеть, анализируя код этого класса. Такой список является двусвязным списком, элементы которого – списки, построенные на массивах:

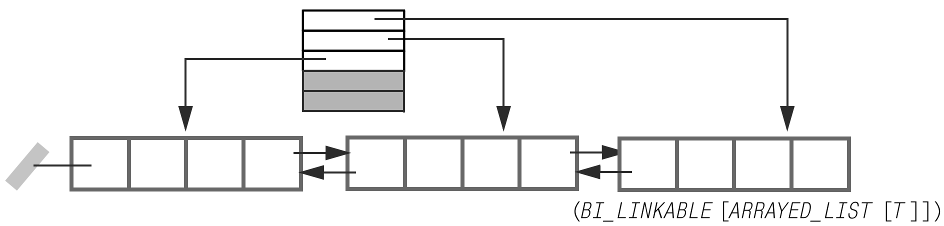


Рис. 13.25. Мультимассивный список

Одним из преимуществ такой структуры является то, что ее никогда не нужно перестраивать. Когда достигается предельная емкость, то создается новый список, построенный на массивах, никакие старые элементы трогать не приходится. В худшем случае эффективность равна O(n) для некоторых ключевых операций, но чаще всего она остается практически приемлемой.

13.9. Хеш-таблицы

Массивы представляют структуры, индексированные целыми числами. Что, если нам нужны другие виды ключей? Строки являются типичным примером. Нам могут понадобиться контейнеры, в которых критерием доступа является строка символов, такие как:

- каталог персон – контейнер, в котором каждый объект содержит информацию о персоне. Вам необходимо получать информацию, задавая имя персоны;
- коллекция веб-страниц, сопровождаемая поисковой машиной; страницы индексируются всеми ключевыми словами, появляющимися на этой странице.

Предположим на минуту, что в первом случае все персоны, собранные в каталоге, имеют имена, отличающиеся первой буквой: Annie, Bertrand, Caroline ... Тогда можно было бы ис-

пользовать массив из 26 элементов, где индекс соответствовал бы коду буквы: 1 – для A, 2 – для B и так далее.

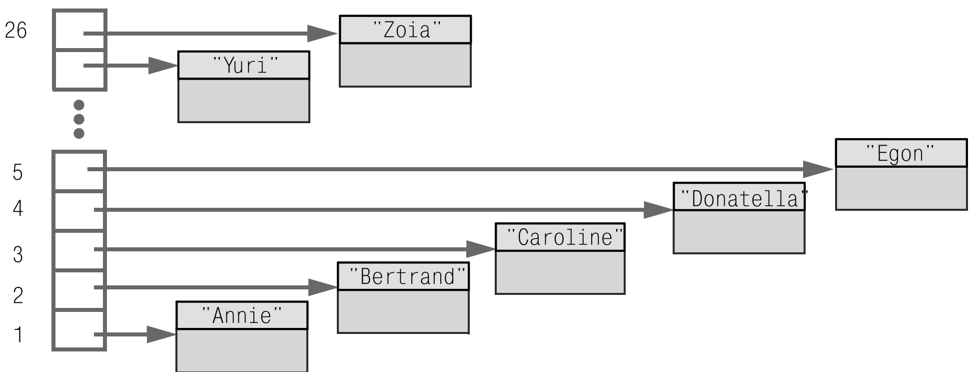


Рис. 13.26. Совершенный хеш

Мы хешировали ключи (строки, представляющие имена) в целые числа из интервала 1...26. «Хеширование» понимается здесь по аналогии с приготовлением котлет – мясо пропускается через мясорубку и разделяется на порции. Более точно:

Определение: хеш-функция

Хеш-функцией на множестве *K* возможных ключей называется функция *h*, которая отображает *K* в некоторый целочисленный интервал *a*...*b*.

Другими словами, для любого $key \in K$ функция дает значение $i = h(key)$, такое, что $a \leqslant i \leqslant b$.

На практике обычно интервал задается в форме 0... *capacity* – 1 для некоторого целого *capacity*. Хеш-функция *h*(*key*) задается в форме *f*(*key*) **Mod**(*capacity*) (по модулю емкости контейнера), где функция *f* возвращает целочисленное значение, приводимое к нужному интервалу взятием по модулю. Массив, применяемый для хранения данных, имеет размерность *capacity*.

В нашем примере используется примитивная хеш-функция, возвращающая порядок в алфавите первой буквы имени. Слегка более сложной хеш-функцией является функция, которая суммирует все ASCII-коды символов, входящих в имя, а затем возвращает остаток при целочисленном делении полученного значения на емкость контейнера – *capacity*.

Хеш-функция зависит только от ключей, а не от числа элементов, так что если *count* – это размерность нашей задачи, то время, затрачиваемое на вычисление функции, есть O(1) или O(*l*), если учитывается длина ключа – *l*, но можно предположить, что хеш-функция использует только первые *K* символов ключа, где *K* – константа.

Предположение, что в нашем примере все имена различаются по первой букве, приводит к тому, что хеш-функция для различных имен дает различные значения. В общем случае хеш-функция называется **совершенной**, если для разных значений ключа она вырабатывает разные значения. Для совершенной хеш-функции вставка и поиск требуют O(1) времени.

В большинстве случаев мы не можем получить совершенную хеш-функцию, даже с описанной выше функцией, вычисляющей сумму кодов всех символов. Для несовершенных

функций встречаются коллизии, когда разные ключи дают одно и то же значение функции. Хорошая хеш-функция характеризуется небольшим числом коллизий. Можно сказать, что функция, вычисляющая сумму кодов с последующим приведением по модулю емкости контейнера, лучше, чем функция, учитывающая только первую букву ключа. Для первой функции коллизии начнут фактически возникать, когда число элементов будет превосходить емкость контейнера *capacity.* Реализация хеш-функций должна уметь справляться с коллизиями.

Одним из методов является так называемое **открытое хеширование**, когда массив комбинируется со связным списком. На последнем приведенном рисунке с совершенным хешированием массив непосредственно содержит все элементы и мог быть объявлен как

```
ARRAY[G]
```

При открытом хешировании мы могли бы использовать массив, элементами которого были бы связные списки:

```
ARRAY[LINKED_LIST[G]]
```

Каждый элемент массива с индексом *i* представляет список объектов, для которых хеш-функция дает значение *i*:

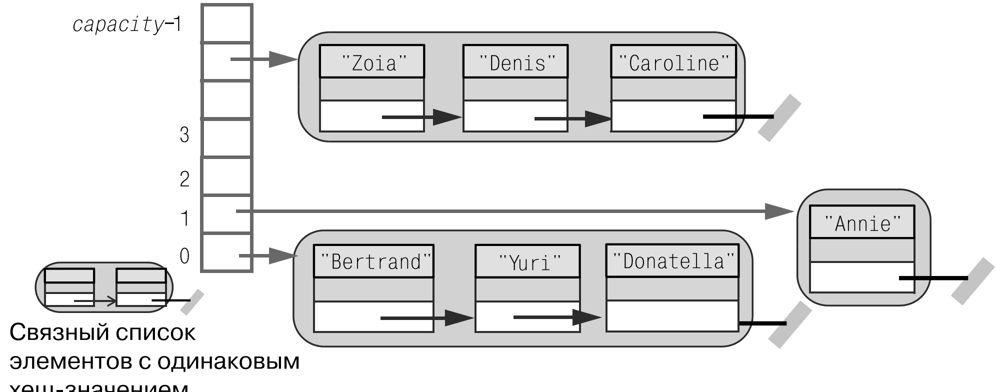


Рис. 13.27. Открытое хеширование, использующее массив связных списков

При поиске или вставке элемента в хеш-таблицу с открытым хешированием первым делом ключ преобразуется в индекс, дающий вход в список, а затем производится последовательный просмотр списка. Первая операция имеет стоимость O(1), а вторая – O(*c*), где *c* – фактор коллизии – среднее число ключей, хешируемых на данный индекс. Если емкость массива *capacity* считать константой, то значение *c* для больших *count* и хорошо распределенной хеш-функции будет O(*count*/ *capacity*), а с учетом нашего предположения – O(*count*). Чтобы избежать линейной зависимости, необходимо периодически перестраивать массив, но тогда лучше использовать другую технику, называемую **закрытым хешированием**.

Закрытое хеширование, применяемое в классе *HASH_TABLE* библиотеки EiffelBase, не использует связных списков, а работает с массивом *ARRAY*[*G*]. В любой момент времени некоторые его позиции заняты, а некоторые – свободны:

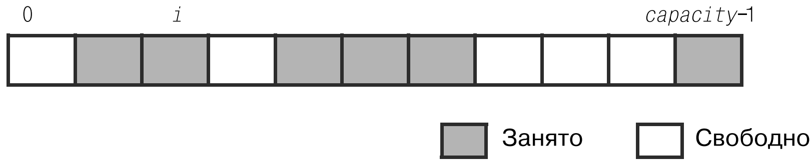


Рис. 13.28. Массив, реализующий хеш-таблицу при закрытом хешировании

Если при вставке хеш-функция вырабатывает уже занятую позицию, например, *i*, как показано на следующем рисунке, то применяемый механизм последовательно будет испытывать другие позиции – *i*1, *i*2, *i*3, пока не найдет свободную ячейку:

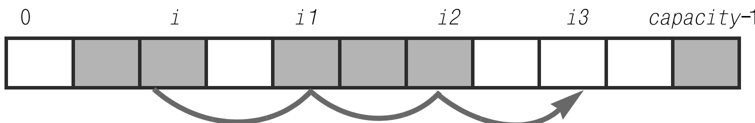


Рис. 13.29. Поиск свободной ячейки

Общий прием состоит в следующем: если хеш-функция вырабатывает позицию для первого кандидата *i* = *f*(*key*) **Mod**(*capacity*), то последующие позиции определяются как i + *increment*, *i* +2 * *increment*, *i* +3 * *increment* и так далее, все по модулю *capacity.* Величина *increment* вычисляется как *f*(*key*) **Mod** (*capacity* **-1**). Такой алгоритм используется в классе *HASH_TABLE* библиотеки EiffelBase (смотри метод *search_for_insertion* для изучения деталей).

Гарантирование завершения процесса поиска означает, что цикл имеет вариант и алгоритм всегда способен найти пустую ячейку. Это достигается подходящим подбором параметров и политикой перестройки массива при его заполнении. Фактически, мы не ждем до последней минуты, – перераспределение начинается, когда коэффициент заполнения достигает граничного значения – 80% в классе *HASH_TABLE.*

Поразительно, но такая политика в сочетании с хорошим выбором хеш-функции приводит к тому, что практически вставка и поиск требуют O(1) затрат (смотри ссылку в конце этой главы на теоретический анализ сложности).

Такое поведение означает, что для практических целей хеш-таблицы почти так же хороши, как и массивы, но позволяют иметь произвольные ключи, так что хеш-таблицу с элементами, идентифицируемыми строковым ключом, можно рассматривать как массив, индексируемый строками, а не целыми.

> Это замечательный результат, так как реальное индексирование строками приводило бы к излишне большим структурам. Рассмотрим, например, ключи, заданные строкой из 7 символов. Число возможных значений ключа равно 26^7, что примерно дает 8 миллиардов значений. Даже если не учитывать проблемы с памятью, было бы абсурдно воспринимать всерьез такие массивы, когда на практике приходится иметь дело с множествами существенно меньшего размера. При хешировании памяти выделяется чуть больше, чем фактически необходимо, но вместе с тем достигается поведение, сравнимое с поведением массива.

Нахождение хеш-функций, приводящих к такому эффективному поведению, является в некотором роде искусством. Образцом и источником вашего вдохновения может послужить функция, используемая в классе *HASH_TABLE.*

Класс *HASH_TABLE*[*G*, *KEY*] является первым примером, где появляются два родовых параметра типа, а не один, как было ранее: *G* задает тип элементов, а *KEY* – тип ключей этих элементов. Этот класс можно использовать, например, для хранения объектов, которые представляют персоны, идентифицируемые именами:

```
personnel_directory: HASH_TABLE [PERSON, STRING ]
```

У этого класса есть несколько фундаментальных методов. Класс имеет единственную процедуру создания *make*. Для создания хеш-таблицы можно применить вызов:

```
create personnel_directory.make (initial_size)
```

Здесь *initial_size* – это некоторое положительное целое. Не имеет большого значения, каким его выбрать. Как следует из его названия, это просто некоторая подсказка для начального выделения памяти. Если вы зададите число много ниже реальной потребности, то это приведет во время выполнения к нескольким дополнительным перестройкам массива.

Рассмотрим запросы, существующие в классе. Чтобы узнать, есть ли в классе элемент с заданным ключом, используйте запрос

```
has (k: KEY ): BOOLEAN
```

Для получения элемента, ассоциированного с заданным ключом, если таковой есть:

```
item (k: KEY ) alias "[]": G assign put
          – Элемент, ассоциированный с заданным ключом, если таковой есть,
          – в противном случае - значение по умолчанию для типа G
      ensure
          default_value_if_not_present:
             not (has (k)) implies (Result = computed_default_value)
```

Постусловие показывает, что если нет элемента с заданным ключом, то результатом является значение по умолчанию типа *G* (ноль для целых, false – для булевских, void – для ссылок). Это не лучший способ тестирования наличия элемента в таблице, так как там может существовать элемент, имеющий значение по умолчанию, так что предварительно ст*о*ит использовать запрос *has* в таких ситуациях.

Спецификация **alias "[]"** показывает, что так же, как и для элементов массива, возможно применение квадратных скобок для элементов хеш-таблиц, что позволяет писать:

```
personnel_directory ["Isabelle"]
```

Эта запись является синонимом:

```
personnel_directory.item ("Isabelle")
```

Форма с квадратными скобками короче и привычнее, так что она будет использоваться в дальнейшем.

Для вставки элемента в таблицу нужно задать как сам элемент, так и его ключ:

```
personnel_directory.put (that_person, "Isabelle")                [8]
```

Это справедливо и тогда, когда ключ является атрибутом элемента:

```
personnel_directory.put (that_person, that_person.name)
```

Класс предлагает четыре операции вставки с одной и той же сигнатурой:

```
put (new: G; k: KEY )      – Команда-присваиватель для элемента.
forse (new: G; k: KEY )
extend (new: G; k: KEY )
       require
          not_present: not has (k)
replace (new: G; k: KEY )
```

Среди них *extend* имеет предусловие, устанавливающее применимость только тогда, когда элемента с заданным ключом нет в таблице; остальные три всегда применимы. Предложение «**note**» в начале класса объясняет, когда следует использовать тот или иной вариант. Я воспроизведу его здесь, опуская некоторые детали.

Варианты вставки в хеш-таблицы (из текста класса HASH_TABLE)

- Используйте *put*, если вы хотите, чтобы вставка происходила только тогда, когда в таблице нет элемента с данным ключом, в противном случае ничего делаться не будет.
- Используйте *force*, если вы хотите делать вставку в любом случае. Это означает, что существующий элемент с данным ключом будет удален.
- Используйте *extend*, если вы уверены, что в таблице нет элемента с заданным ключом, – это обеспечит более быструю вставку.
- Используйте *replace*, если вы хотите заменить существующий элемент с заданным ключом, ничего не делая в противном случае.

В первых двух случаях процедура будет устанавливать значение булевского запроса *found*, позволяющего узнать после вставки, был ли уже в таблице элемент с заданным ключом.

Объявление элемента с заданием псевдонима и команды-присваивателя выглядит так:

```
item (k: KEY ) alias "[]": G assign put
```

В результате доступна скобочная нотация, позволяющая вставлять элементы в хеш-таблицу, применяя инструкцию, подобную присваиванию:

```
personnel_directory ["Isabelle"]:= that_person
```

Фактически, это краткая форма записи вызова *put*, более простая, чем рассмотренная в **[8].**

Для удаления элемента с заданным ключом используйте:

```
remove (k: KEY )
```

Команда не имеет эффекта, если элемента с заданным ключом в таблице нет. Выяснить, что фактически происходило, может запрос *removed*.

Для удаления всех элементов служит процедура *clear_all*.

Выполняя эти операции, нет необходимости заботиться о размере структуры данных. Благодаря перестраиваемым массивам Eiffel, сами методы заботятся о выделении достаточного пространства для всех текущих элементов.

Если вы явно хотите изменить размер, то следует вызвать метод *accommodate*(*n*:*INTEGER*), который добавит в таблицу новые ячейки, не меняя уже существующие.

Вот обзор стоимости операций хеш-таблицы.

Операция	**Метод класса** *HASH_TABLE*	**Сложность**
Доступ по ключу	*item, has*	O(1)
Вставка по ключу	*put, force, extend*	O(*count*)
Замена по ключу	*replace*	O(1)
Удаление по ключу	*remove*	O(1)

При работе с большими системами с большим числом объектов вы обнаружите, что хеш-таблицы станут одним из ваших любимых инструментов.

13.10. Распределители

Массивы и хеш-таблицы являются индексируемыми структурами.

- При вставке элемента необходимо задать некоторую идентифицирующую информацию: индекс – для массивов, ключ – для таблиц.
- При доступе необходимо указать ассоциированный индекс элемента или его ключ.

Структуры, к изучению которых мы приступаем, следуют другой политике. Они не используют ключей или другой идентифицирующей информации. Вы просто вставляете элемент, применяя для этого обычную процедуру:

```
put (x: G)
        -- Добавить x в текущую структуру.
```

Сравните с *put*(*x:G, i:INTEGER*) для массивов или с *put*(*x:G, k:KEY*) для хеш-таблиц. Когда же приходится получать элемент, у вас нет возможности его выбора. Вы делаете запрос

```
item: G
        -- Элемент, полученный из текущей структуры
    require
        not is_empty
```

У запроса нет аргументов (сравните с запросом *item*(*i:INTEGER*)*:G* для массивов или с *item*(*k:KEY*)*:G* для хеш-таблиц). Мы называем такие структуры **распределителями** по аналогии с автоматом, выдающим банки с напитком. Автомат, а не покупатель, решает, какую банку выдать покупателю.

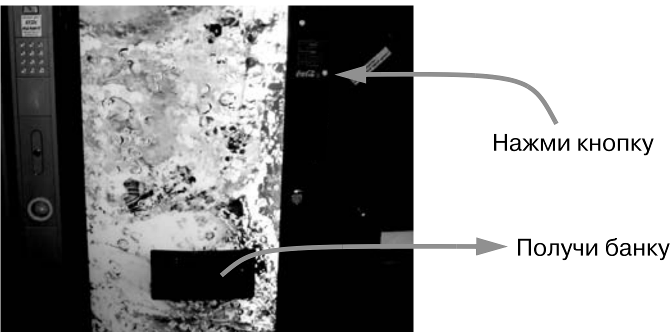


Рис. 13.30. Торговый автомат

Распределители отличаются политикой, используемой для выбора выдаваемого элемента.

- Last-In First-Out: выбирается элемент, поступивший последним из существующих. Распределитель с политикой LIFO называется **стеком**.
- First-In First-Out: выбирается элемент, поступивший первым из существующих. Распределитель с политикой FIFO называется **очередью**.
- Для **очереди с приоритетами** элементы обладают приоритетами (целое или вещественное число). Тогда по запросу будет выдаваться элемент, обладающий наибольшим приоритетом среди присутствующих. Может показаться, что этот случай ближе к индексированным структурам, но все же это пример распределителя, поскольку приоритет – это внутреннее свойство элемента, и распределитель, а не пользователь выбирает, какой элемент будет выдан.

У всех распределителей существуют четыре базисных метода: *put* и *item* с сигнатурами и предусловиями, показанными выше, а также булевский запрос

```
is_empty: BOOLEAN
          – Правда, что элементов нет?)
и команда для удаления элемента:
remove
          – Удалить элемент из текущей структуры.
       require
          not is_empty
```

Точно так же, как *item* не позволяет выбирать получаемый элемент, *remove* не позволяет выбирать удаляемый элемент. Удаляется тот элемент, который можно получить по запросу *item*, если выполнить его непосредственно перед вызовом *remove*.

Хорошая реализация распределителей должна выполнять все эти операции за время O(1). Примеры вскоре будут даны.

В некоторых библиотеках можно найти операции, которые комбинируют эффект *item* и *remove:* функцию, скажем, *get*, которая удаляет элемент, а в качестве результата выдает удаленный элемент. Такую функцию можно реализовать в терминах *item* и *remove*:

```
get: G
          – Функция с побочным эффектом, нарушающая принципы методологии!
       do
```

```
    Result:= item
    remove
end
```

Мы не будем использовать такие функции, так как они меняют структуру и возвращают результат, нарушая правило, что только команды, но не запросы, могут менять состояние структуры (принцип разделения команд и запросов). По причинам, объясненным в предыдущих главах, предпочтительнее позволять клиентам получать доступ и удалять элементы двумя разными методами – запросом, свободным от побочного эффекта, и командой.

В следующих двух разделах рассматриваются стеки и очереди. Мы не будем рассматривать очереди с приоритетами, но всегда можно обратиться к библиотеке EiffelBase и ознакомиться с классом *PRIORITY_QUEUE*.

13.11. Стеки

Стек – это распределитель с политикой LIFO: элемент, к которому можно получить доступ, есть элемент, поступивший последним из существующих в распределителе. Этот элемент располагается в «вершине» стека, что соответствует естественному образу стека в обыденном смысле этого термина. Примером может служить множество словарей, громоздящихся на моем столе в предположении, что первым я могу взять словарь, находящийся на вершине этой груды (стека).

Рис. 13.31. Стек

«Ханойская башня», которая изучается в следующей главе, посвященной рекурсии, также дает пример работы со стеком.

Стек. Основы

Операции над стеком часто известны как:

- **Push** (втолкнуть элемент на вершину стека – команда *put*);
- **Pop** (вытолкнуть элемент с вершины – команда *remove*);
- доступ к элементу вершины (запрос *item*).

Эти операции можно визуализировать.

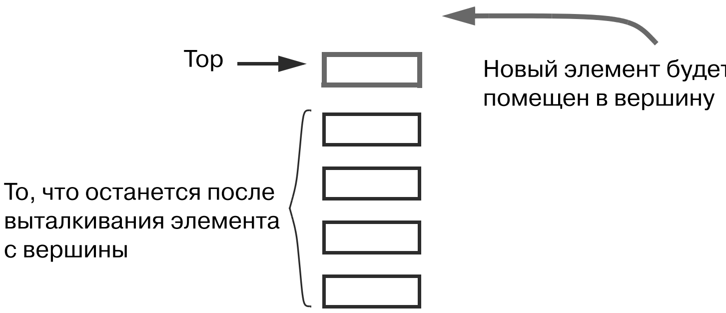


Рис. 13.32. Концептуальный образ стека

Использование стеков

Стеки имеют множество применений в компьютерной науке. Два примера из реализации языка программирования: один – статический (разбор, иллюстрируемый в простейшем случае обработкой «польской нотации»), другой – динамический, управление вызовами программ в период выполнения.

Предположим, что вы хотите вычислить математическое выражение в «польской нотации» – форме, часто применяемой в калькуляторах, а иногда и во внутреннем представлении компиляторов и интерпретаторов. Преимущество этой нотации в том, что устраняется неопределенность порядка вычислений без использования скобок – каждый знак операции применим к операндам, непосредственно предшествующим знаку. Результат операции над операндами является операндом следующей операции.

Рассмотрим для примера выражение

```
2 + (a + b) * (c - d)
```

В польской записи оно выглядит так

```
2 a b + c d - * +
```

Как будет происходить вычисление этого выражения? Первым знаком операции является +, так что выполнится сложение операндов a и b, предшествующих плюсу. Затем выполнится операция вычитания, затем умножение двух вычисленных операндов, последним выполнится сложение полученного результата с константой 2. Для простоты все операции бинарны, но схема легко адаптируется на произвольную «-арность» операций.

Следующий алгоритм, использующий стек операндов s, вычисляет общее выражение в польской записи с бинарными операциями:

```
from                                    -- Инициализация пуста
until
            "Все термы выражения уже прочитаны"
loop
```

```
    "Чтение очередного терма выражения - x"
  if "x является операндом" then
    s.put (x)
  else — x является знаком бинарной операции
          — Получить два верхних операнда
      op1:= s.item; s.remove
      op2:= s.item; s.remove
        — Применить операцию к операндам и поместить результат в стек:
      s.put (application (x, op1, op2))
  end
end
```

В алгоритме используются две локальные переменные *op1* и *op2*, представляющие операнды. Функция *application* вычисляет результат применения бинарной операции к ее операндам, например, *application*('+', 2, 3) возвращает значение 5. На следующем рисунке показана ключевая операция алгоритма, соответствующая предложению **else**, – обрабатывается знак умножения для выражения нашего примера.

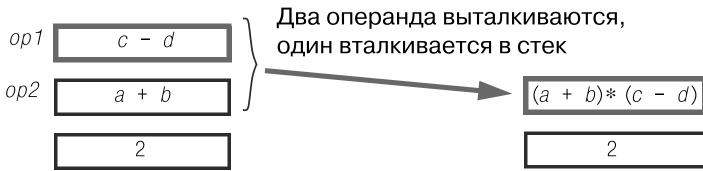


Рис. 13.33. Вычисление выражения, записанного в польской нотации

Корректная реализация алгоритма должна справляться с ошибочным вводом (проверяя *s.is_empty* перед вызовом *item* и *remove* и проверяя, что *x* является знаком операции); необходимо также предусматривать возможность операций различной «-арности».

Наш второй пример лежит в основе поддержки исполняемой среды при реализации каждого современного языка программирования и присутствует в каждой операционной системе (это, конечно, сильное утверждение, но ни один контрпример не приходит на ум). Рассмотрим язык программирования, позволяющий методу вызывать другой метод, который, в свою очередь, может вызывать метод, и ситуация может повторяться. В результате появляется цепочка вызовов:

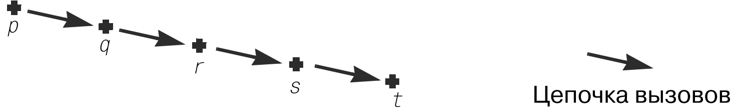


Рис. 13.34. Вызов метода

В любой момент времени в период выполнения несколько методов – от *p* до *t* на рисунке – были вызваны, начали свою работу, но еще ее не завершили. Последний вызванный метод в этом случае называется **текущим методом.** Рассмотрим одну из его команд, например, присваивание $x := y + z$. Если только *x, y, z* не являются атрибутами охватывающего класса, то

они должны принадлежать текущему методу и быть либо его **аргументами** (но не *x*, поскольку аргументу нельзя присваивать значения), либо **локальными переменными**. Будем использовать термин «локальные» для обеих категорий. Для выполнения операторов программы, таких как присваивание, код, генерируемый компилятором, должен иметь доступ ко всем локальным переменным. Решением этой проблемы является создание для каждого вызова метода активирующей записи, содержащей его локальные переменные:

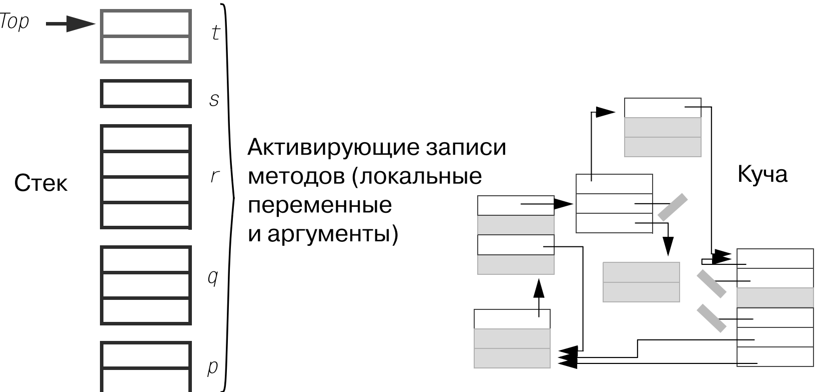


Рис. 13.35. Стек периода выполнения и куча

Структура справа называется «*кучей*», она содержит объекты, получаемые в результате вызова процедуры создания или ее эквивалента. Для нашего обсуждения интерес представляет **стек вызовов**, также называемый стек периода выполнения (чаще всего просто стек), содержащий активирующие записи для всех текущих активных методов. Поскольку метод не может завершиться, пока не завершатся методы, вызовы которых он инициировал, и стартовавшие позже него, подходящей схемой активации является стратегия LIFO и стек будет подходящей структурой.

> Во многих языках программирования тексты методов могут быть *гнездованы* – вложены друг в друга. Тогда операторы могут ссылаться не только на локальные переменные текущего метода, но и на локальные переменные любого охватывающего блока. Это означает, что выполнению может понадобиться доступ не только к верхней активирующей записи, но и к некоторым другим, расположенным ниже ее. В этой схеме структура, представляющая активирующие записи, по-прежнему называется стеком, но использует расширенное понятие стека. В языке Eiffel нет необходимости в гнездовании методов[1].

В момент вызова метода механизм создает новую активизационную запись с локальными переменными метода, инициализированными значениями по умолчанию, и аргументами метода, которые инициализируются значениями фактических аргументов, переданных в

[1] В языке Eiffel локальные переменные могут быть объявлены только на уровне метода. У метода нет внутренних блоков, в которых могут объявляться локальные переменные блока, например, внутри составного оператора или цикла. Блочная структура метода, даже когда нет вложенности методов, требует расширенного понятия стека.

точку вызова. Эта запись размещается в вершине стека. При завершении работы метода запись удаляется из стека и на вершину поднимается следующая в стеке запись.

Преимущество использования стека в том, что записи представляют не различные методы, а только различные сеансы *выполнения*. Как результат, эта техника позволяет поддерживать рекурсивные методы – методы, вызывающие себя непосредственно или косвенно. Создавая новую запись при вызове рекурсивного метода, позволяем каждому вызову хранить множество своих локальных переменных. Рекурсия – тема следующей главы, а ее реализация, основанная главным образом на стеках, – тема отдельного раздела.

Реализация стеков

Как и для некоторых других структур этой главы, существуют две общие категории реализации стеков, основанные на массивах и на связных списках. Наиболее общая реализация использует массив *rep* типа *ARRAY*[*G*] и целочисленную переменную *count* с инвариантом

```
count >= 0 ; count <= rep.capacity
```

Здесь емкость *capacity* представляет число элементов массива (*upper* – *lower* + 1). Для массивов, индексируемых с 1, элементы стека, если они есть, хранятся в позициях от 1 до *count*.

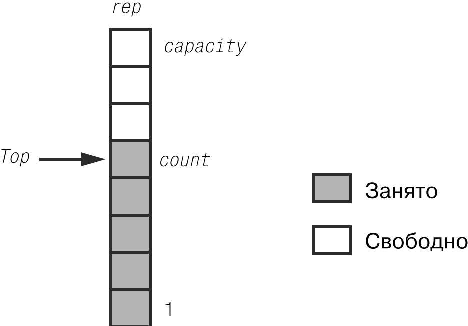


Рис. 13.36. Реализация стека на массиве

> В классе *ARRAY* число элементов массива известно как *count* и как *capacity*, инвариант свидетельствует, что значения этих атрибутов эквивалентны. Не следует путать атрибут *count* для массивов с *count* для стеков – атрибутом, который задает число элементов стека, разворачиваемого на массиве.
> Мы уже сталкивались с этим различием, когда рассматривали список, реализованный на массиве. В обоих случаях реализация построена на массиве, в то время как спецификация задает другой контейнерный тип.

В этой реализации запрос *item*, который дает элемент, расположенный в вершине стека, просто возвращает *rep*[*count*] – элемент массива в позиции *count*. Достаточно просто может быть реализована и команда *remove*: *count*:= *count* -1, а команда *put(x)* – как

```
count:= count + 1                                          [9]
rep.force (x, count)
```

Здесь используется команда *force* для массивов, заставляющая перестроить массив, если отведенной памяти становится недостаточно.

Более подробно с реализацией можно познакомиться, изучая класс *ARRAYED_STACK* из библиотеки EiffelBase (фактически классу не нужен *rep,* поскольку он наследуется от *ARRAY*, но концептуально это эквивалентно, а мы все же формально наследование еще не изучали). Использование *force* в алгоритме для *put* означает, что можно не беспокоиться о размере массива – массив будет создаваться с установками по умолчанию, а потом подстраиваться под нужный размер данных.

Конечно, физическая память компьютера ограничена, но в большинстве случаев ее хватает для наших потребностей.

Перестройка массива, применяемая в Eiffel, не является общедоступной в других программных средах, поэтому там часто стеки, базируемые на массиве, имеют ограниченную емкость. Соответствующий класс есть и в Eiffel – *BOUNDED_STACK*. Для такого стека наряду с *count* используется и запрос *capacity*, и булевский запрос *is_full*, чье значение дается выражением *count* = *capacity.* В этом случае, так же, как существует предусловие для команды *remove*, будет существовать предусловие и для команды *put* – *is_full*. Реализация этой команды для такого стека использует *put* для массива, а не *force*, как в вышеприведенном тексте **[9]**. Выполнение предусловия гарантирует корректность выполнения *put*.

Все рассмотренные выше операции имеют сложность O(1).

Вариантом стека ограниченной емкости, реализованного на массиве, является стек, растущий вниз:

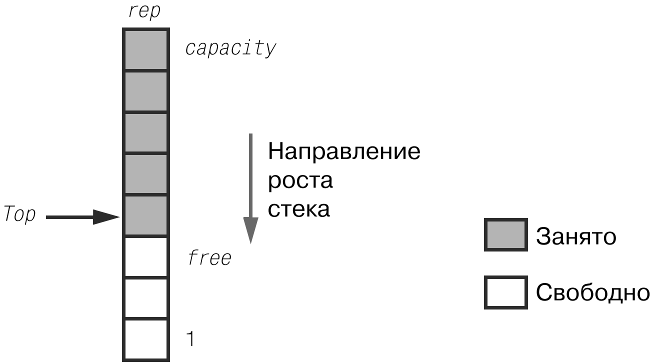


Рис. 13.37. Реализация на массиве стека, растущего вниз

В этом представлении *count* более не является атрибутом, вместо этого появляется скрытый атрибут *free*, задающий индекс первой свободной ячейки. Запрос *count* по-прежнему остается доступным, но реализуется он теперь функцией, возвращающей значение *capacity – free.*

Инвариант теперь устанавливает, что *free* >= *0 и free* <= *capacity.* Сравните этот инвариант с инвариантом для *count* в предыдущем представлении стека.

Случай *free = 0* соответствует *is_full*, а *free* = *capacity* соответствует is_*empty.* Элементы стека, если они есть, располагаются в позициях от *capacity* до *free* +1. Метод *remove* реализуется просто: *free* = *free* +1, а *put* реализуется как

```
rep.force (x, free)
free:= free - 1
```

Если память ограничена и приходится одновременно работать с двумя стеками, то можно оба стека располагать на одном массиве, но на разных его концах; один растет вверх, другой вниз, что отражено на следующем рисунке:

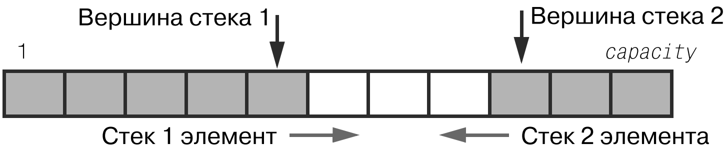


Рис. 13.38. Два стека на одном массиве

Преимущество этого подхода в том, что оптимальным образом используется память, если только оба стека не достигают своего максимума одновременно, поскольку

```
max (count1 + count2) ⩽ max (count1) + max (count2)
```

В упражнении вас попросят написать реализацию класса *TWO_STACK*, воплощающего эту идею.

Наряду с реализацией на массивах вполне допустимо строить стек на связном списке. Действительно, связный список, изученный ранее в этой главе, содержит готовую реализацию стека. Рисунок ниже иллюстрирует этот подход: первая ячейка является вершиной стека, а остальные – телом стека.

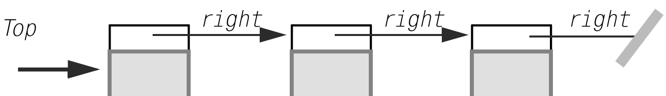


Рис. 13.39. Связный стек

Операция *put*(*x*) реализуется просто как *rep.put_front*(*x*), где rep задает связный список. Аналогично, *item* реализуется как *rep.first* и так далее. Класс *LINKED_STACK* в EiffelBase обеспечивает такую реализацию. Все базисные операции имеют сложность O(1), хотя чуть медленнее, чем их двойники на массивах, например, *put_front* из класса *LINKED_LIST,* а следовательно, и *put* из *LINKED_STACK* должны сперва создать и отвести память ячейке *LINKABLE.*

Все базисные операции над стеком во всех рассмотренных реализациях выполняются за константное время, за исключением, как отмечалось, редкой операции *force* в перестраиваемом массиве, реализующем стек.

Операция	**Метод в классе стека**	**Сложность**	**Комментарий**
Доступ к вершине	*item*	O(1)	
Вталкивание на вершину	*put*	O(1)	При автоматической перестройке иногда O(*count*)
Удаление с вершины	*remove*	O(1)	

13.12. Очереди

Очереди с их политикой FIFO («первый пришел – первый ушел») полезны во многих приложениях. Вот типичные примеры.

- При моделировании, особенно в варианте, известном как моделирование дискретных событий. Программа выполняет шаги, моделируя события, происходящие в некотором процессе – на сборочной линии, собирающей машины из комплектующих деталей, в информационной сети, передающей сообщения, в магазине, обслуживающем покупателей. Часто обработка событий в этих случаях удовлетворяет политике FIFO, и очередь представляет возникающие события в процессе.
- Аналогичное моделирование требуется и при организации графического интерфейса пользователя (GUI), где события инициируются пользователем – щелчки мыши, нажатия клавиш, перемещение курсора – и должны обрабатываться в порядке их возникновения.
- В операционных системах и при организации параллельного программирования часто применима схема *«поставщик – потребитель»*, где один процесс – поставщик – генерирует некоторую информацию, другой – потребитель – читает и обрабатывает ее в порядке поступления. Структура, используемая для обмена информацией, представляет очередь в варианте, называемом «**буфер**».

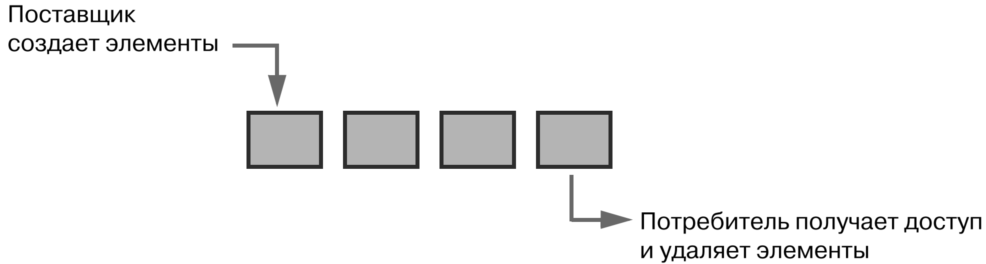


Рис. 13.40. Поставщик – потребитель, взаимодействие через буфер

Последний рисунок может служить концептуальным представлением любой очереди, а не только буфера – элементы поступают с одного конца, а удаляются с другого.

Как и для стеков, реализация может использовать для представления либо массив, либо связный список. В последнем случае очередь выглядит так:

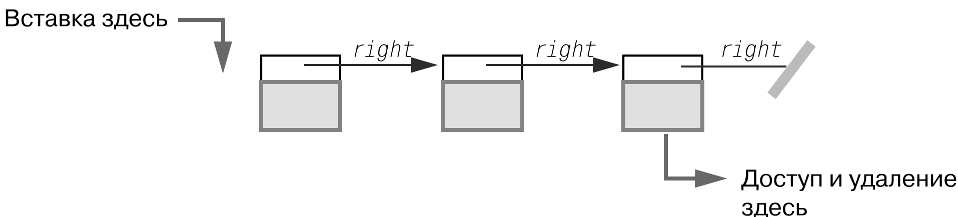


Рис. 13.41. Очередь на связном списке

Операция *put(v)* реализуется просто как *rep.put_front*(*v*) (здесь, как обычно, *rep* задает список). Запрос *item* возвращает последний элемент списка, а *remove* – удаляет его. Класс *LINKED_QUEUE* из EiffelBase сопровождается инвариантом

```
is_always_after: not empty implies rep.after
```

Выполнение инварианта гарантирует, что курсор всегда находится в конце списка.

Представление очереди массивом немного изощреннее, чем для стеков, поскольку необходимо добавлять элементы на одном конце, а удалять на другом. В этом случае требуются два указателя, которые в классе *ARRAYED_ QUEUE* называются *in_index* и *out_index*, и оба являются закрытыми атрибутами. Запрос *count* по-прежнему дает число элементов очереди. Естественным, но не лучшим решением является хранение элементов очереди в интервале *in_index* .. *out_index*, как показано на рисунке:

Рис. 13.42. Возможное состояние до очереди на массиве

Реализация для такого представления очевидна:

```
remove: out_index = out_index + 1,
put(v): rep[in_index]:= v; in_index:= in_index +1
```

При таком подходе память, отведенная массиву, может быть быстро исчерпана при добавлении элементов, даже если происходит их удаление, поскольку пространство в начале массива останется неиспользованным:

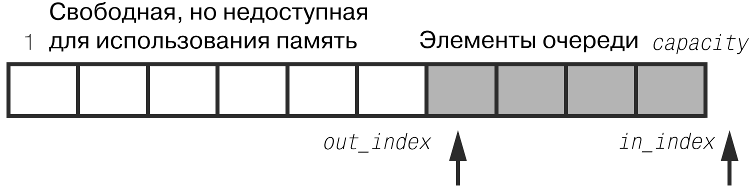


Рис. 13.43. Очередь на массиве в момент достижения правого конца массива

Решение: когда маркер *in_index* превосходит емкость *capacity*, то операция *put* должна по кругу возвращаться в начало массива, аналогично должна вести себя *remove*. Концептуально массив превращается в круг:

Вот пример реализации *put* в классе *ARRAYED_QUEUE*:

```
put (v: G)
            — Добавить v как новый элемент.
```

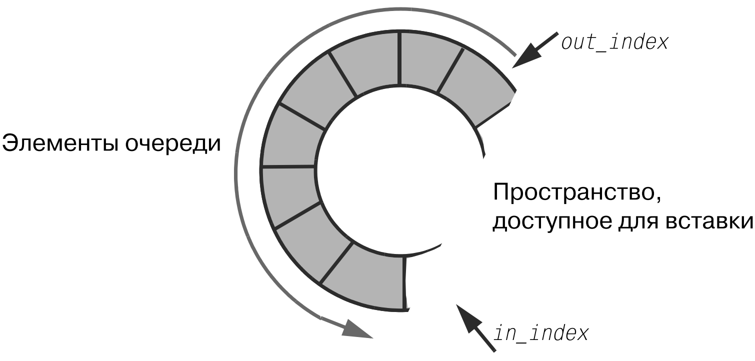


Рис. 13.44. Массив как бублик

```
do
    if count + 1 = rep.count then grow end
    rep [in_index]:= v
    in_index:= (in_index + 1) \\ capacity
    if in_index = 0 then in_index:= capacity end
end
```

Первый оператор выделяет массиву дополнительную память, если ее действительно не хватает. Процедура *grow* просто вызывает *resize* для нашего массива. Увеличение *in_index* выполняется по модулю *capacity* (*i* \\ j дает остаток от деления *i* на *j*, а *i // j* дает целую часть от деления нацело). Реализация настраиваемая (смотри заключительный **if**...), массив *rep* может индексироваться от 1 до *capacity*, но может – рекомендованное упражнение – индексироваться, начиная с нуля.

Очереди при подходящей реализации столь же эффективны, как и стеки.

Операция	**Метод в классе очереди**	**Сложность**	**Комментарий**
Доступ к старейшему элементу	*item*	O(1)	
Добавление элемента	*put*	O(1)	При автоматической перестройке иногда O(*count*)
Удаление старейшего элемента	*remove*	O(1)	

13.13. Итерирование структуры данных

Контейнерные структуры данных, вроде тех, что рассматривались в данной главе, являются хранилищами объектов. Зачастую на таких структурах одно и та же операция применяется поочередно ко всем объектам структуры. Этот процесс называется *итерированием* структуры

данных. В дополнение к этому термину механизм, осуществляющий итерирование, также имеет собственное имя.

Определение: итератор

Итератор – это механизм, который может применять одну или несколько операций к элементам контейнера, рассматривая это как операцию над структурой в целом.

Мы уже видели многие примеры итерирования контейнерных структур. Все они соответствуют общему образцу: если *your_list* относится к типу *LINKED_LIST* [*T*] или к более общему типу *LIST* [*T*] (для любой реализации списка) и есть процедура

```
some_opereation(x: T)
```

то следующая схема итерирует операцию над списком

```
from
        your_list.start
invariant
        — Все операции перед курсором уже подверглись операции some_operation
until
        your_list.after
loop
        some_operation (your_list.item)
        your_list.forth
variant
        your_list.count - your_list.index + 1
end
```

Как альтернатива, операция применяется только к элементам, которые удовлетворяют условию, заданному функцией *your_condition (x:T): BOOLEAN*. В этом случае тело цикла следует изменить следующим образом:

```
if your_condition (your_list.item) then
        some_operation (your_list.item)
end
```

Другие варианты итерирования включают:

- применение операции ко всем элемента контейнера, пока не встретится элемент, удовлетворяющий (не удовлетворяющий) некоторому условию;
- выяснение, существует ли по крайней мере один элемент (все элементы), удовлетворяющий некоторому условию.

Выделение общих схем является правильной стратегией. Еще лучше создать повторно используемый код, чтобы не приходилось каждый раз писать его заново. И на самом деле, на Eiffel можно применять итерационный механизм без написания циклов, используя такие методы, как *do_all* и *do_if*, которые применимы ко всем классам, задающим списки. Они раз и навсегда охватывают все предшествующие циклические структуры, так что два последних примера можно записать гораздо проще:

```
your_list.do_all (agent your_operation)
your_list.do_if (agent your_operation, agent your_condition)
```

Здесь **agent** *your_operation* обозначает объект, который представляет процедуру *your_operation*, готовую к применению к каждому элементу, а **agent** *your_condition* аналогично задает запрос. Для понимания деталей следует дождаться рассмотрения агентов в последующих главах.

13.14. Другие структуры

Структуры данных, рассмотренные нами, относятся к наиболее важным в программировании, но они далеко не единственные. Мы уже упоминали о деревьях, и в следующих главах поговорим о них подробнее. Обобщением деревьев является полезное во многих приложениях, например, в сетях, понятие графа, ориентированного или неориентированного, а также понятие мультиграфа. В разделе литературы предлагаются книги, посвященные фундаментальным структурам данных, обычно в сочетании с фундаментальными алгоритмами. Кроме того, приводятся учебники, поддерживающие курс «Структуры данных и алгоритмы», который включен в обязательную программу большинства университетов, обучающих информатике.

13.15. Дальнейшее чтение

Рис. 13.45. Дональд Кнут (2005)

Рис. 13.46. Альфред Ахо (2007)

Дональд Кнут: «Искусство программирования», т 1. «Основные алгоритмы», т 3. «Сортировка и Поиск», М., Мир. 1976 г. (Последнее издание в России – 2008 г.)

Широко известный учебник по структурам данных и алгоритмам. Часть из задуманного 7-томного выпуска, из которого вышли в печать три тома (некоторые главы четвертого известны в виде отдельных выпусков).

1. Ахо А., Хопкрофт Дж., Ульман Дж. «Построение и анализ вычислительных алгоритмов», М., Мир, 1976 г.

Компактный обзор наиболее важных алгоритмов и структур данных. До сих пор остается великолепным обзором в этой области.

Кормен Т., Лейзерсон Ч., Ривест Р. «Алгоритмы: построение и анализ», 2002 г.

Великолепный современный учебник.

2. Bertrand Meyer «Reusable Software», Prentice Hall, 1994.

Изложение принципов проектирования, применяемых при построении качественных, повторно используемых библиотек; сопровождается примерами из EiffelBase.

13.16. Ключевые концепции этой главы

- Статическая типизация делает программы более ясными и позволяет обнаруживать многие ошибки на этапе компиляции.
- Универсальный класс имеет один или несколько родовых параметров, представляющих типы. Это обеспечивает гибкость и, в частности, полезно при описании контейнерных структур.
- Структуры данных должны поддерживать перестройку, позволяющую настраивать размер в зависимости от объема приходящих данных.
- Для согласованности библиотек желательна политика стандартного именования методов.
- Абстрактная сложность позволяет оценить производительность алгоритмов вне зависимости от выбора «железа», фокусируясь на поведении алгоритма для данных больших размеров и игнорируя аддитивные и мультипликативные константные множители.
- Нотация «О-большое», как в $O(n^2)$, выражает абстрактную сложность.
- Массивы обеспечивают доступ и замену элементов за константное время благодаря индексам из фиксированного интервала. Хотя перестройка размера массива возможна, они не подходят в случаях частой вставки или удаления элементов.
- Хеш-таблицы обобщают массивы, позволяя вместо целочисленных индексов использовать почти произвольные ключи, например строки, сохраняя при этом доступ и замену элементов в основном за константное время.
- Списки описывают последовательные структуры, и в варианте со ссылками поддерживают быстрые операции вставки и удаления.
- Распределители позволяют вам получать доступ, вставку и удаление элементов в строго определенном месте. Политика LIFO («последний пришел – первый ушел») управляет стеками, FIFO («первый пришел – первый ушел») – очередями.
- Стеки, в частности, полезны для представления вложенных структур и интенсивно используются в компиляторах и операционных системах. Реализация стека массивом является общепринятой; на одном массиве возможно размещение двух стеков.
- Очереди особенно полезны при моделировании и в параллельном программировании, где известны как буферы. При реализации очереди массивом последний должен рассматриваться как закольцованный.

Новый словарь

Abstract complexity	Абстрактная сложность	**Activation record**	Активизационная запись
Actual generic parameter	Фактический родовой параметр	**Array**	Массив
		Call chain	Цепочка вызовов
Complexity	Сложность	**Correctness**	Корректность
Cursor	Курсор	**Dispenser**	Распределитель
Dynamic typing	Динамическая типизация	**FIFO**	Первый пришел – первый ушел

Formal generic parameter	Формальный родовой параметр	**Generic class**	Универсальный (родовой) класс
Generic derivation	Родовое порождение	**Genericity**	Универсальность
Hash table	Хеш-таблица	**Heap**	Куча
Linked list	Связный (односвязный) список	**LIFO**	Последний пришел – первый ушел
List	Список	**Parameter**	Параметр
Priority queue	Очередь с приоритетами	**Queue**	Очередь
		Run-time stack	Стек периода выполнения
Stack	Стек		
Static typing	Статическая типизация	**Validity**	Правильность

13-У. Упражнения

13-У-1. Словарь

Дайте точные определения всем терминам словаря.

13-У-2. Карта концепций

Добавьте новые термины в карту концепций, построенную в предыдущих главах.

13-У-3. Два в одном

Напишите класс *DOUBLE_STACK* [*G*], реализующий два стека на одном массиве. Вы можете назвать соответствующие методы *put1*, *put2*, *remove1*, *remove2* и так далее. Стеки имеют ограниченный размер, так что позаботьтесь включить правильные предусловия и инварианты класса.

13-У-4. Индексация, начинающаяся с нуля

Реализация, которую мы рассматривали для очередей, построенных на массиве, подобна классу *ARRAYED_QUEUE* из библиотеки EiffelBase, но без наследования, использует массив *rep*, индексируемый начиная с единицы.

1. Используя как образец реализацию *put*, данную в тексте, напишите процедуру *remove* и процедуру создания *make*, задающую пустую очередь.
2. Перепишите все три метода, используя индексацию массива, начинающуюся с нуля.

13-У-5. Обращение списка

Напишите процедуру обращения списка для двусвязного списка и списка, построенного на массиве, поместив их в класс, наследующий от соответствующих классов EiffelBase: *TWO_WAY_LIST* или *ARRAYED_LIST*.

14
Рекурсия и деревья

Рис. 14.1.

Смеющаяся корова, изображенная на фирменном жетоне «Смеющаяся Корова», носит в качестве сережек фирменные жетоны, на которых, я подозреваю, но зрение не позволяет убедиться в правильности моего предположения, изображена корова с фирменными жетонами, на которых изображена ... (надеюсь, идея понятна).[1]

Эта реклама, появившаяся в 1921 году, все еще хорошо работает, являясь примером структуры, определенной *рекурсивно* в следующем смысле:

Рекурсивное определение

Определение понятия является рекурсивным, если оно включает один или более экземпляров самого понятия.

«Рекурсия» — использование рекурсивного определения — широко применяется в программировании: она позволяет элегантно определять *синтаксические структуры*; мы также познакомимся с рекурсивно определенными *структурами данных* и рекурсивными *процедурами*.

Мы будем использовать термин «*рекурсивный*» как сокращение «рекурсивно определенный» — рекурсивная грамматика, рекурсивная структура данных. Но это только соглашение, поскольку нельзя сказать, что понятие или структура сами по себе рекурсивны. Все, что мы знаем, — это то, что их можно рекурсивно описать в соответствии с вышеприведенным оп-

[1] У нас известен рекурсивный стишок, вошедший в поговорку: «У попа была собака, он ее любил. Она съела кусок мяса, он ее убил, и в землю закопал, и на могиле написал, что у попа была собака ...»

ределением. Любое частичное понятие – в том числе структура, задающая Смеющуюся корову, – может быть определено как рекурсивно, так и без использования рекурсии.

При рассмотрении свойств рекурсивно определенного понятия будем применять рекурсивные *доказательства* — обобщающие индуктивные доказательства, использующие целые числа для индуктивного шага.

Рекурсия является прямой, если определение *A* ссылается на экземпляр *A*, и косвенной, если для 1 <= *i* < n (для некоторого *n* >=2) определение каждого A_i ссылается на A_{i+1}, а определение A_n ссылается на A_1.

В этой главе нас будут интересовать такие понятия, для которых рекурсивные определения естественны, элегантны и удобны. Примеры будут включать рекурсивные программы, рекурсивные синтаксические определения, рекурсивные структуры данных. Мы также получим некоторое представление о рекурсивных доказательствах.

Один из классов рекурсивных структур данных – *деревья* в их различных представлениях – появляются во многих приложениях и отражают очень точно идею рекурсии. В этой главе рассматривается важный случай *бинарных* деревьев.

14.1. Основные примеры

В этот момент могут возникнуть вполне обоснованные сомнения – а есть ли смысл в рекурсивных определениях, как можно определить понятие через само понятие, «масло масляное»?

Вы вправе сомневаться. Не все рекурсивные определения хороши для определения чего-либо. Когда вас просят дать характеристику кому-нибудь, а вы отвечаете: «Света? Ну, это просто Света, что еще можно сказать!» – то вы не много нового сказали. Так что следует позаботиться о критериях, гарантирующих полезность определения, даже если оно рекурсивно.

Прежде чем мы это сделаем, позвольте убедиться прагматичным путем, ознакомившись с несколькими типичными примерами, когда рекурсия очевидно полезна и осмысленна. Это придаст нам твердую убежденность, бо́льшую, чем просто вера, основанная на надеждах и молитвах, что рекурсия — это практически полезный способ определять грамматики, структуры данных и алгоритмы. После чего настанет время для подходящего математического обоснования рекурсивных определений.

Рекурсивные определения

С введением универсальности мы получаем возможность определять тип как:

T1 класс, не являющийся универсальным, такой как *INTEGER* или *STATION*;

T2 родовое порождение в форме *C*[*T*], где *C* — универсальный класс, а *T* – тип.

Это определение рекурсивно, оно просто означает, что, при наличии универсальных классов *ARRAY* и *LIST*, правильными классами также будут:

- *INTEGER, STATION* и им подобные в соответствии с определением T1;
- согласно случаю T2, прямые родовые порождения: *ARRAY*[*INTEGER*]*, LIST*[*STATION*] и так далее.

Снова рекурсивно применяя T2: *ARRAY*[*LIST*[*INTEGER*]], *ARRAY*[*ARRAY* [*LIST*[*STATION*]]] и так далее – родовые порождения любого уровня вложенности.

Используя подобный прием, можно дать ответ на упражнение из первой главы, где требовалось дать определение алфавитному (лексикографическому) порядку.

Рекурсивно определенные грамматики

Рассмотрим подмножество Eiffel с двумя видами операторов.

- Присваивание, в его обычной форме *variable:= expression*, рассматриваемое здесь как терминальное и далее не уточняемое.
- Условный оператор, имеющий только часть **then** (без **else**) для простоты.

Грамматика, определяющая язык, такова:

```
Оператор  ≜  Присваивание | Условный
Условный  ≜  if Условие then Оператор end
```

Для наших непосредственных целей будем полагать, что «Условие» является терминальным понятием. Это определение грамматики очевидно рекурсивно, поскольку определение «Оператор» включает «Условный», а его определение, в свою очередь, включает «Оператор». Но так как здесь присутствует нерекурсивная часть определения — «Присваивание», то в целом грамматика четко определяет правильные конструкции языка:

- просто Присваивание;
- Условный, содержащий Присваивание: **if** *c* **then** *a* **end**;
- то же самое с произвольной степенью вложенности: **if** *c*1 **then if** *c*2 **then** *a* **end end**, **if** *c*1 **then if** *c*2 **then if** *c*3 **then** *a* **end end end** и так далее.

Рекурсивные грамматики на самом деле являются незаменимым средством для описания любого языка, который, как все распространенные языки, поддерживает вложенные структуры.

Рекурсивно определенные структуры данных

Класс *STOP* представляет понятие остановки для линии метро:

```
class STOP create
…
feature
    next: STOP
            — Следующая остановка на той же линии.
        …Другие компоненты, здесь опущенные (см.6.5)
end
```

Наивная интерпретация будет предполагать, что каждый экземпляр *STOP* содержит экземпляр *STOP*, который, в свою очередь, содержит остановку и так до бесконечности, как в схеме со Смеющейся коровой. Это и в самом деле было бы так, если бы *STOP* принадлежал развернутым, а не ссылочным типам:

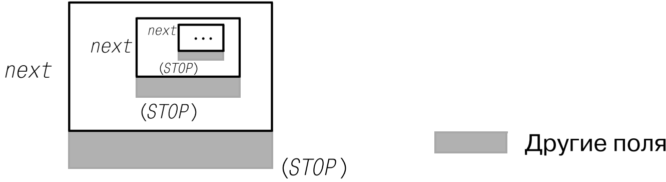


Рис. 14.2. Вложенные поля (интерпретация не корректна)

Такое попросту невозможно. Но *STOP* в любом случае является ссылочным типом, подобно любому классу, определенному как **class** *X*... без всяких других квалификаций, так что реальная картина выглядит так:

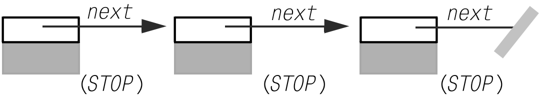


Рис. 14.3. Линия, связанная ссылками

Рекурсия в таком определении структуры данных просто указывает, что каждый экземпляр класса потенциально содержит ссылку на экземпляр того же класса, «потенциально» – поскольку ссылка может иметь значение void, и тогда действие рекурсии прекращается, как для последней остановки на рисунке.

В той же 6-й главе, где рассматривались линии метро, шла речь и о классе *PERSON* с атрибутом *spouse* типа *PERSON*.

Это весьма общая ситуация в определении полезных структур данных, начиная от связных списков до деревьев различного вида (таких как бинарные деревья, изучаемые позже в этой главе). Определения полезных классов часто включают ссылки на объекты определяемого класса или (косвенная рекурсия) на классы, зависящие от определяемого.

Рекурсивно определяемые алгоритмы и программы

Известная последовательность чисел Фибоначчи обладает многими прекрасными свойствами и появляется во многих приложениях математики и естественных наук. Она имеет следующее определение:

```
F_0 = 0
F_1 = 1
F_i = F_{i-1} + F_{i-2}        -- Для i > 1
```

Почувствуй историю

Кролики Фибоначчи

Леонардо Фибоначчи из Пизы (1170 – 1250) сыграл ключевую роль в знакомстве Запада с трудами индийских и арабских математиков. Он известен также и собственными исследованиями, лежащими в основании современной математики. Он сформулировал задачу, приводящую к его знаменитой последовательности (которая была известна еще индийским математикам):

Человек получил пару кроликов и поместил их в загон, окруженный со всех сторон стеной. Как много пар кроликов можно получить от этой пары в год, если каждый месяц каждая пара производит новую пару, которая становится продуктивной на втором месяце?

Решение дает следующее рассуждение. Пары кроликов в месяце *i* включают пары кроликов, уже существующие в предыдущем месяце (кролики не умирают); обозначим их число как F_{i-1}, плюс потомство, принесенное кроликами, жившими в месяце *i*-2 (кролики, появившиеся в месяце *i*-1, потомства не приносят). Это и дает приведенную выше формулу, созда-

Рис. 14.4. Фибоначчи

ющую последовательность целых чисел 0, 1, 1, 2, 3, 5, 8 и так далее. Формула приводит к рекурсивной программе, вычисляющей F_n для любого n:

```
fibonacci (n: INTEGER): INTEGER
        -- Элемент с индексом n в последовательности Фибоначчи.
    require
        non_negative: n >= 0
    do
        if n = 0 then
            Result := 0
        elseif n = 1 then
            Result := 1
        else
            Result := fibonacci(n - 1)+ fibonacci(n - 2)
        end
    end
```

Время программирования!

Рекурсивная версия Фибоначчи

Напишите небольшую программную систему, включающую вышеприведенную рекурсивную функцию и печатающую ее результат. Протестируйте ее для небольших значений *n*, включая 12, как в оригинальном тексте Фибоначчи.

Функция включает два рекурсивных вызова. То, что это работает, может казаться несколько загадочным (вот почему стоит проверить выполнение функции на нескольких значениях). После изучения этой главы легитимность таких рекурсивно определенных программ должна стать вполне понятной.

Принципиальным аргументом в пользу такого способа записи программы есть то, что она вполне соответствует математическому определению последовательности Фибоначчи. Дальнейшее рассмотрение покажет, что этот факт не столь впечатляющий, поскольку получить нерекурсивную версию также не сложно.

Время программирования!

Нерекурсивная версия Фибоначчи
Можете ли вы, не заглядывая в дальнейший текст, написать функцию, вычисляющую N-е число Фибоначчи, используя цикл, а не рекурсию?

Следующая функция дает тот же результат, что и рекурсивная версия. Проверьте это на нескольких значениях.

```
fibonacci1 (n: INTEGER): INTEGER
        -- Элемент с индексом n в последовательности Фибоначчи.
        -- (Нерекурсивная версия.)
    require
        positive: n >=1
    local
        i, previous, second_previous: INTEGER
    do
        from
            i := 1 ; Result := 1
        invariant
            Result = fibonacci(i )
            previous = fibonacci (i - 1)
        until i = n loop
            i := i + 1
            second_previous := previous
            previous := Result
            Result := previous + second_previous
        variant
            n - i
        end
    end
```

Для удобства в этой версии предполагается, что *n* >= 1, а не *n* >= 0. Благодаря правилам инициализации *previous* начинается с 0, что гарантирует начальное выполнение инварианта, так как $F_0 = 0$. Переменная *second_previous* обновляется на каждом шаге цикла и ей не нужно специальной инициализации.

Эта версия более удалена от оригинального математического определения, но все же проста и понятна. Заметьте: инвариант цикла ссылается для удобства на рекурсивную функцию, представляющую официальное математическое определение. Некоторые могут предпочитать рекурсивную версию в любом случае, но это дело вкуса. В зависимости от компилятора рекурсивная версия может быть менее эффективной по времени выполнения.

Оставим в стороне вкус и эффективность. Если бы мы рассматривали только такие примеры, то необходимость рекурсивных программ была бы далеко не очевидной. Нам необходимы примеры, в которых рекурсия обеспечила бесспорные преимущества, например, за счет того, что ее нерекурсивный аналог был бы значительно сложнее и труднее в понимании. Такие примеры существуют в большом количестве. Одним из них является очаровательная головоломка – «Ханойская башня». В этом примере сконцентрировано много полезных свойств рекурсии, и практически нет не относящихся к делу деталей.

14.2. Ханойская башня

В величественном храме Бенареса, под куполом, отмечающим центр мира, на медной плите установлены три бриллиантовых стержня. Каждый стержень высотой в локоть и тонок, как пчелиная талия. На один из этих стержней в начале мироздания Бог поместил 64 диска из чистого золота. Самый большой диск покоится на медной плите, а остальные, друг друга меньше, создают пирамиду, вздымающуюся к вершине стержня. Это и есть священная башня Брахмы.

Ночью и днем неустанно священники, сменяя друг друга, работают, чтобы перенести священную башню на третий бриллиантовый стержень, не нарушив при этом священных правил, установленных Брахмой. Когда их работа будет закончена, падет башня, падут брахманы и наступит конец мира.

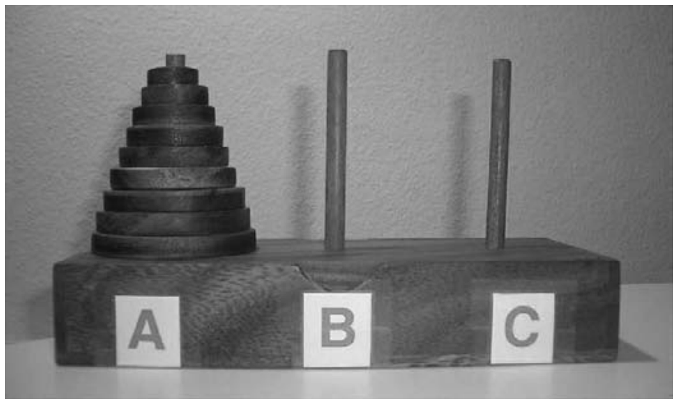


Рис. 14.5. Башня Ханоя (должна быть башней Бенареса?) из 9 дисков в начальном состоянии

Несмотря на восточный орнамент, эта история является созданием французского математика Эдуарда Лукаса (подписывающегося как «N. Claus de Siam» — анаграмма «Lucas d'Amiens», с добавлением названия его родного города). На рынке в Таиланде (Сиам) я купил подобную башню, показанную на рисунке. Метки **A, B, C** — это мое добавление. Не буду распространяться на тему, почему я выбрал модель, сделанную из дерева, а не из бриллиантов, золота и меди. Но вполне законно спросить, почему на ней только 9 дисков, хотя портфель у меня был большой и мог бы вместить башню из 64 дисков.

Время теста!

Размер Ханойской башни

Почему коммерчески доступные модели Ханойской башни имеют размеры много меньшие, чем 64 диска?

(Подсказка: игра сопровождается бумажным свертком, на котором дается решение головоломки в форме последовательности ходов; **A** => **C**, **A** => **B** и т. д.)

Чтобы ответить на этот вопрос, давайте оценим минимальное число ходов H_n (ход — это перемещение диска с одного стержня на другой), требуемое для решения задачи. Если задача имеет решение, то нужно перенести *n* дисков со стержня **A** на стержень **B**, используя стержень

C как промежуточный. При этом нужно соблюдать правило Будды, запрещающее класть больший диск на меньший. В оригинальной версии $n = 64$, для небольшой модели $n = 9$.

Заметим, что для любой стратегии перемещения в некоторый момент необходимо перенести самый большой диск со стержня **A** на стержень **B**, а это возможно лишь при условии, что все остальные n -1 дисков в этом момент находятся на диске **C** в требуемом порядке:

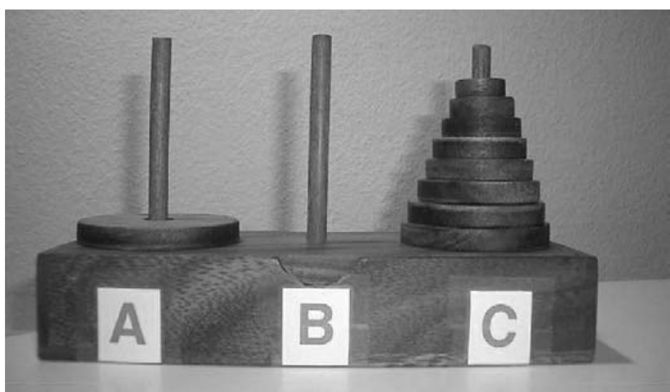


Рис. 14.6. Промежуточное состояние

Каково минимальное число ходов, необходимое для достижения этого промежуточного состояния? Необходимо перенести n -1 диск со стержня **A** на стержень **C**, не перемещая самый большой диск и используя стержень **B** как промежуточный. Ввиду симметричности задачи для этого потребуется H_{n-1} ходов.

Сразу же по достижении этого состояния можно перенести за один ход самый большой диск с **A** на **B**, после чего перенести $n - 1$ диск с **C** на **B**, используя **A** как промежуточный. Общее количество ходов:

$$H_n = 2 * H_{n-1} + 1$$

Учитывая, что H_0 равно 0, получим:

$$H_n = 2^n - 1$$

Как следствие, нетрудно получить ответ на наш тест. Вспомним, что $2^{10} = 1024$, или примерно 10^3, и получим, что число ходов 2^{64} примерно равно $1{,}5 * 10^{19}$.

Год – это примерно 30 миллионов секунд. Если предположить, что священники Бенареса за секунду выполняют один ход – весьма неплохая скорость для переноса золотого диска, – всю работу они закончат за 500 миллиардов лет, что примерно в 30 раз превосходит оценочный возраст существования нашей вселенной. Даже компьютеру, выполняющему 100 миллионов ходов за секунду, при моделировании этой задачи для переноса дисков потребуется не одна тысяча лет.

Вывод оценки H_n был *конструктивным*, в том смысле, что он дает *практическую стратегию* для перемещения дисков.

- Переместить $n - 1$ диск с **A** на **C**, используя **B** как промежуточное хранилище и руководствуясь правилами игры.

- После этого стержень **B** будет пуст, а на **A** будет находиться только один самый большой диск, который и переносится за один ход с **A** на **B**. Этот ход соответствует всем правилам игры – перенос одного диска с вершины одного стержня на другой стержень, в вершине которого нет диска, размер коего меньше размера переносимого диска.
- После этого переместить $n - 1$ диск с **C** на **B**, используя **A** как промежуточное хранилище, руководствуясь правилами игры. Самый большой диск, находящийся на **B**, не мешает переносу дисков меньшего размера.

Эта стратегия превращает число ходов $H_n = 2^n - 1$ из теоретического минимума в практически достижимую цель. Мы можем записать алгоритм в виде рекурсивной процедуры, входящей в класс *NEEDLES*:

```
hanoi (n: INTEGER; source, target, other: CHARACTER)
        - Перенос n дисков из source на target,
        - используя other как промежуточное хранилище,
        - в соответствии с правилами игры "Ханойская башня"
    require
        non_negative: n >= 0
        different1: source /= target
        different2: target /= other
        different3: source /= other
    do
        if n > 0 then
            hanoi (n-1, source, other, target)
            move (source, target)
            hanoi (n-1, other, target, source)
        end
    end
```

Обсуждение контрактов для рекурсивных методов добавит дополнительные предложения в предусловие и постусловие.

По соглашению стержни представляются символами – 'A', 'B', 'C'. Другое соглашение, принятое в этой главе (уже использованное в предыдущих примерах), состоит в подсветке рекурсивных ветвей кода, – процедура *hanoi* содержит два таких участка.

Базисная операция *move*(*source*, *target*) перемещает один диск с вершины стержня *source* на вершину *target*. Предусловие устанавливает, что на *source* должен находиться, по крайней мере, один диск, а на *target* диска либо нет, либо диск в вершине имеет больший размер, чем перемещаемый диск. Запишем *move* как процедуру, выводящую на консоль инструкцию по перемещению диска:

```
move (source, target: CHARACTER)
        - Инструкция по перемещению диска с source на target.
    do
        io.put_character (source)
        io.put_string (" to ")
        io.put_character (target)
        io.put_new_line
    end
```

Время программирования!

Ханойская башня

Напишите систему с корневым классом NEEDLES, включающим процедуры *hanoi* и move. Проверьте их работоспособность на примерах.

Например, выполните вызов

```
hanoi (4, 'A', 'B', 'C')
```

В результате должна быть напечатана последовательность из пятнадцати ($2^4 - 1$) ходов:

A на *C*	*B* на *C*	*B* на *A*
A на *B*	*A* на *C*	*C* на *B*
C на *B*	*A* на *B*	*A* на *C*
A на *C*	*C* на *B*	*A* на *B*
B на *A*	*C* на *A*	*C* на *B*

Эта последовательность ходов успешно переносит диски с **A** на **B** в полном соответствии с правилами игры.

Один из способов анализа рекурсивного решения – процедуры *hanoi* – состоит в том, чтобы рассматривать перемещение $n - 1$ дисков, как один обобщенный ход. В этом случае мы могли бы начать перемещение с этого хода (с **A** на **C**):

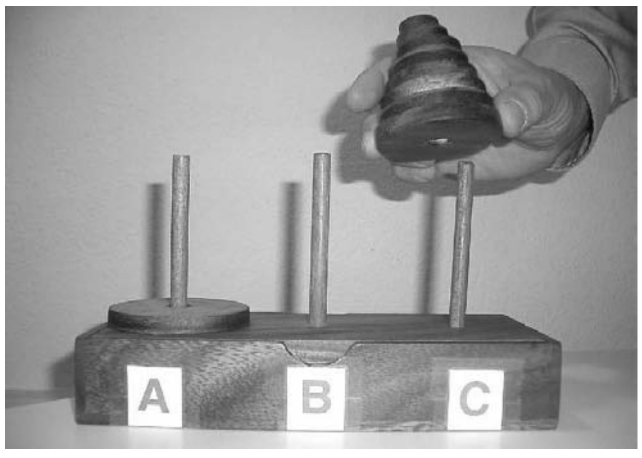


Рис. 14.7. Начальный глобальный ход Фибоначчи

После этого можно сделать обычный ход, перенеся самый большой диск на целевой стержень, а затем снова выполнить обобщенный ход, который легален, поскольку все диски обобщенного хода меньше, чем диск, уже лежащий на целевом стержне.

Конечно, это фиктивное рассмотрение, поскольку правилами разрешается перенос только одного диска за один ход, но перемещая $n - 1$ диск, можно применять ту же технику, зная, что целевой стержень либо пуст, либо содержит диски большего размера. Так можно рекур-

сивно выполнять перемещения, уменьшая каждый раз значение *n*. Когда же *n* становится равным нулю, то делать ничего не нужно.

Не стоит легкомысленно относиться к примеру с Ханойской башней. Это решение служит прекрасной моделью для многих рекурсивных программ с важными практическими приложениями. Простота алгоритма является результатом использования двух рекурсивных вызовов, и это делает эту задачу идеальной для изучения свойств рекурсивных алгоритмов, к чему мы вернемся чуть позже в этой главе.

14.3. Рекурсия как стратегия решения задач

В предыдущих главах мы рассматривали управляющие структуры языка программирования как технику, используемую для решения сложных задач.

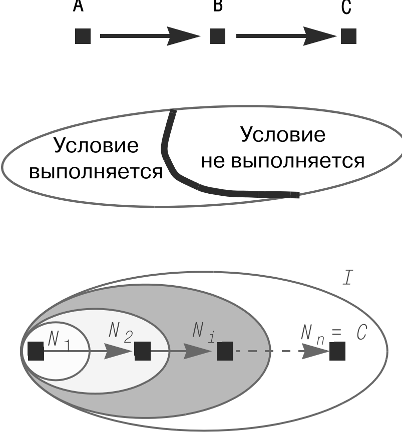


Рис. 14.8.

- **Составной** оператор (последовательность). Семантику последовательности можно выразить так: «*Я знаю кого-то, кто может провести меня от текущей точки до B, и знаю того, кто может провести от B до C, так что можно просить их, работая последовательно, провести меня до C*».
- **Условный** оператор означает: «*Я знаю кого-то, кто может решить задачу в одном случае, и знаю того, кто может решить ее для всех других возможных случаев, что позволяет мне просить их работать в зависимости от возникающей ситуации*».
- Решение, использующее **цикл**, можно выразить так: «*Я не знаю, как добраться до C, но я знаю область I* (*инвариант*)*, содержащую C, знаю кого-то, кто может привести меня в эту область* (*инициализация*)*, и знаю того* (*тело цикла*)*, кто может приблизить меня к C, при условии, что я нахожусь в области I. Расстояние до цели будет уменьшаться* (*вариант*) *таким образом, что за конечное число шагов я достигну требуемой мне окрестности C. Мне остается попросить моего первого друга привести меня в область I, а затем просить второго друга приближать меня к C, пока я не достигну цели*».
- **Процедура**, как способ решения задачи, означает: «Я знаю кого-то, кто может решать эту задачу в общем случае, так что мне нужно лишь сформулировать мою специальную задачу в его терминах и попросить решить задачу для меня».

Что можно сказать о рекурсивном решении? К кому нужно обращаться? Ответ – к себе! Возможно – несколько раз (как в случае с Ханойской башней и многих других)!

Зачем обращаться к другим, если я доверяю себе (по крайней мере, я так думаю)?

Теперь мы знаем, что эта стратегия не так глупа, как может казаться с первого раза. Я прошу себя решить ту же задачу, но на подмножестве исходных данных или на нескольких таких подмножествах. Тогда я могу справиться с задачей, если удастся частные решения объединить в решение задачи в целом.

Такова идея рекурсии, рассматриваемая как стратегия решения сложной задачи. Она связана с некоторыми предыдущими стратегиями.

- Рекурсия предполагает процедурную стратегию, так как она основана на решении той же задачи.
- Она использует свойства циклической стратегии: оба подхода приближают решение полной проблемы, покрывая решение расширяющегося множества данных. Но дает более общий подход, так как на каждом шаге может комбинироваться несколько частных решений. Позже мы детально сравним стратегии цикла и рекурсии.

14.4. Бинарные деревья

Если Ханойская башня является квинтэссенцией рекурсивной процедуры, то бинарные деревья являются квинтэссенцией рекурсивных структур данных. Их можно определить следующим образом:

Определение: бинарное дерево

Бинарное дерево над *G*, для произвольного типа данных *G*, задается конечным множеством элементов, называемых узлами, каждый из которых содержит значение типа *G*. Узлы, если они есть, разделяются на три непересекающихся подмножества:

- единственный узел, называемый корнем бинарного дерева;
- (рекурсивно) два бинарных дерева над *G*, называемых левым поддеревом и правым поддеревом.

Все это просто выразить в каркасе класса, не включающем методов:

```
class BINARY_TREE [G] feature
item: G
left, right: BINARY_TREE[G]
end
```

Ссылка void указывает на пустое дерево. Проиллюстрируем бинарное дерево над целыми (рис. 14.9).

Форма «Ветвления» – это наиболее общий стиль представления бинарных деревьев, но не единственный: возможно представление в форме вложенности, которое для данного примера выглядит так (рис. 14.10).

Определение бинарного дерева явно допускает, что дерево может быть пустым. Без этого, конечно, рекурсивное определение приводило бы к бесконечной структуре, в то время как бинарные деревья, что также предписано определением, являются конечными структурами данных.

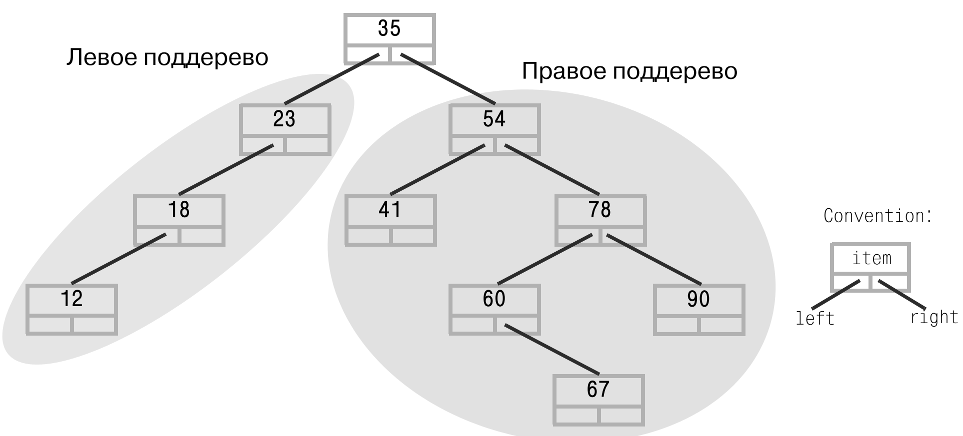


Рис. 14.9. Соглашение: Бинарное дерево (представленное «ветвлением»)

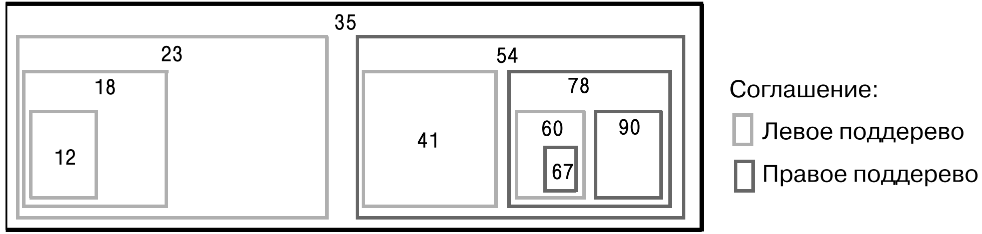


Рис. 14.10. Бинарное дерево (вложенное представление)

Если бинарное дерево не пусто, то оно всегда имеет корень и может не иметь поддеревьев, иметь только левое или только правое поддерево или иметь оба поддерева.

Любой узел бинарного дерева сам рассматривается как бинарное дерево. Достаточно взглянуть на два последних рисунка. Узел, помеченный как 35, задает полное дерево, 23 – его левое поддерево, 54 – правое. Узел 78 задает корень дерева, которое является правым поддеревом правого поддерева полного дерева. Это позволяет говорить о правом и левом поддереве каждого узла. Эту ассоциацию можно сделать формальной, дав другой пример рекурсивного определения.

Определение: дерево, ассоциированное с узлом

Любой узел *n* бинарного дерева *B* определяет бинарное дерево B_n следующим образом:

- если *n* – корень *B*, то B_n – это просто *B*;
- в противном случае из предыдущего определения следует, что *n* – это одно из поддеревьев *B*. Если *B*' – это поддерево, то определим B_n как B'_n (узел, связанный с *n*, рекурсивно, в соответствующем поддереве).

Рекурсивные процедуры над рекурсивными структурами данных

Большинство методов класса, определяющего рекурсивную структуру класса, будут строиться рекурсивно с учетом рекурсивного определения данных. Простым примером является метод, подсчитывающий число узлов бинарного дерева. В пустом дереве число узлов равно нулю, для непустого дерева число узлов равно сумме трех значений: для корня и числа узлов соответственно левого и правого поддеревьев. Нетрудно записать это наблюдение в виде рекурсивной функции, входящей в класс *BINARY_TREE*.

```
count: INTEGER
        -- Число узлов.
    do
        Result := 1
        if left /= Void then Result := Result + left.count end
        if right /= Void then Result := Result + right.count end
    end
```

Заметьте схожесть этой функции с процедурой *Hanoi*.

Дети и родители

Дети (непосредственные потомки) узла – сами узлы – являются корневыми узлами левого и правого поддеревьев:

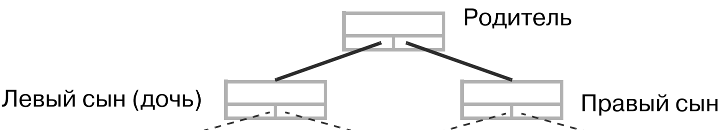


Рис. 14.11. Бинарное дерево (представление «ветвлением»)

Если *C* – сыновний (дочерний) узел *B*, то *B* – **родитель** *C*. Более точно мы можем сказать, что *B* является «родителем» *C*, благодаря следующему результату:

Теорема: «Единственный родитель»

Каждый узел бинарного дерева имеет в точности одного родителя, за исключением корня, у которого нет родителей.

Теорема кажется очевидной, но мы докажем ее, что даст нам возможность познакомиться с *рекурсивными доказательствами*.

Рекурсивные доказательства

Рекурсивное доказательство теоремы о единственном родителе в значительной степени отражает рекурсивное определение бинарного дерева.

Если бинарное дерево *BT* пусто, то теорема выполняется. В противном случае бинарное дерево имеет корень и два непересекающихся бинарных дерева, о которых мы можем предположить – «рекурсивная предпосылка», – что они оба удовлетворяют теореме. Это следует

из определений «бинарного дерева», «ребенка» и «родителя», так что узел *C* может иметь родителя *P* в *BT* только одним из трех возможных случаев:

P1 *P* является корнем *BT*, а *C* является корнем либо левого, либо правого поддерева;

P2 они оба принадлежат левому поддереву, и *P* является родителем *C* в этом поддереве;

P3 они оба принадлежат правому поддереву, и *P* является родителем *C* в этом поддереве.

В случае P1 узел *C* по гипотезе рекурсивности, являясь корнем, не имеет родителей в своем поддереве, так что у него есть единственный родитель – корень всего дерева *BT*. В случаях P2 и P3, опять-таки по гипотезе рекурсивности, *P* был единственным родителем *C* в соответствующем поддереве, и это остается верным и во всем дереве.

Любой узел *C*, отличный от корня, удовлетворяет одному из трех рассмотренных вариантов и, следовательно, имеет в точности одного родителя. Только если *C* является корнем дерева, он не будет соответствовать рассмотренным ситуациям и, как следствие, у него не будет родителей, что и завершает доказательство теоремы.

Подобные рекурсивные доказательства полезны, когда необходимо установить, что некоторое свойство выполняется для всех экземпляров рекурсивно определенного понятия. Структура доказательства определяется структурой определения.

- Для любой нерекурсивной ветви определения необходимо доказать свойство непосредственно (в примере нерекурсивной ветвью является пустое дерево).
- Для рекурсивной ветви определяется новый экземпляр понятия в терминах существующих экземпляров. Для них можно предположить, что свойство выполняется (это и есть «гипотеза рекурсивности»), после чего требуется доказать, что при этих предположениях свойство выполняется.

Эта схема применима в целом для всех рекурсивно определяемых понятий. Мы увидим ее применение к рекурсивно определенной процедуре *hanoi*.

Бинарные деревья выполнения

Интересный пример бинарного дерева можно получить, моделируя выполнение процедуры *hanoi*, например, для трех дисков. Каждый узел содержит аргументы данного вызова, а левое и правое поддерево соответствуют рекурсивным вызовам:

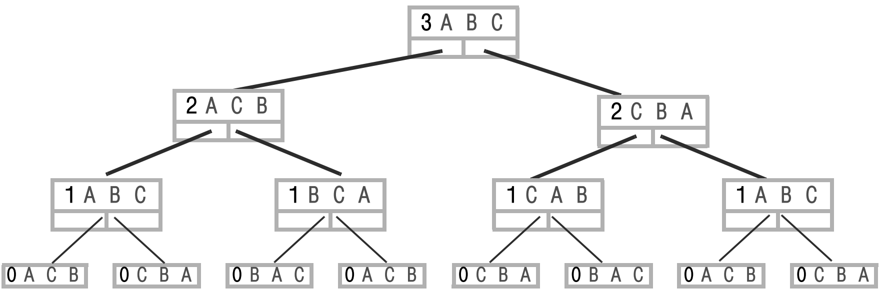


Рис. 14.12. Выполнение Hanoi, рассматриваемое как бинарное дерево

Добавление операции *move* позволило бы реконструировать последовательность операций. Формально мы выполним это позднее.

Этот пример высвечивает связь межу рекурсивными алгоритмами и рекурсивными структурами данных. Для методов, число рекурсивных вызовов в которых задавалось переменной,

а не равнялось двум, как в *hanoi*, выполнение моделировалось бы не бинарным деревом, а деревом общего вида.

Еще о свойствах бинарных деревьях и о терминологии

Как отмечалось, узел бинарного дерева может иметь:

- как левого, так и правого сына, подобно узлу 35 из нашего примера;
- только левого сына, подобно всем узлам левого поддерева, помеченным значениями 23, 18, 12;
- только правого сына, подобно узлу 60;
- не иметь сыновних узлов. Такие узлы называются **листьями** дерева; в примере листьями являются узлы с пометками 12, 41, 67 и 90.

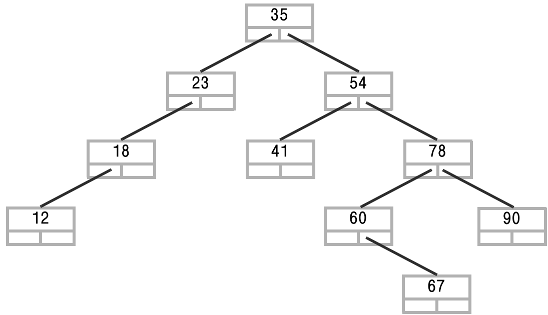


Рис. 14.13. Копия ранее приведенного дерева

Определим восходящий путь в бинарном дереве как последовательность из нуля или более узлов, где любой узел последовательности является родителем предыдущего узла, если таковой имеется. В нашем примере узлы с метками 60, 78, 54 формируют восходящий путь. Справедливо следующее свойство, являющееся следствием теоремы о единственном родителе.

Теорема: «Путь к корню»

Из любого узла бинарного дерева существует единственный восходящий путь, заканчивающийся корнем дерева.

Доказательство. Рассмотрим произвольный узел *C* бинарного дерева. Построим восходящий путь, начинающийся в *C*. В соответствии с теоремой о единственном родителе такой путь определяется единственным образом. Если путь конечен, то заканчиваться он должен в корне дерева, поскольку любой другой узел имеет родителя, и следовательно, построение восходящего пути могло бы быть продолжено. Для завершения доказательства необходимо показать, что все пути конечны. Единственный способ построения бесконечного пути (учитывая, что число узлов бинарного дерева по определению конечно) состоит в том, что путь включает цикл. Если некоторый узел *n* встретится дважды, то он встретится сколь угодно много раз, так что бесконечный путь должен содержать подпоследовательность в форме *n*... *n*. Но это означает, что *n* появляется в своем собственном левом или правом поддереве, что невозможно по определению бинарных деревьев.

Рассмотрение нисходящих путей позволяет установить следующий факт, как следствие предыдущей теоремы.

Теорема: «Нисходящий путь»

Для любого узла бинарного дерева существует единственный нисходящий путь от корня дерева к узлу, проходящий последовательно через левую или правую связь.

Весом бинарного дерева является максимальное число узлов среди всех нисходящих путей от корня к листьям дерева. В примере вес бинарного дерева равен 5, он достигается на пути, который ведет от корня к листу, помеченному как 67.

Это понятие можно определить рекурсивно, следуя снова рекурсивной структуре определения. Вес пустого дерева равен нулю. Вес непустого дерева равен 1 плюс максимум (рекурсивно) из весов левого и правого поддеревьев. Мы можем добавить соответствующую функцию в класс *BINARY_TREE*:

```
height: INTEGER
        — Максимальное число узлов нисходящего пути.
    local
        lh, rh: INTEGER
    do
        if left /= Void then lh := left.height end
        if right /= Void then rh := right.height end
        Result := 1 + lh.max (rh)
    end
```

Здесь рекурсивное определение адаптируется к соглашению, принятому для класса, который рассматривает только непустые поддеревья. Отметьте опять-таки схожесть с *hanoi*.

Операции над бинарными деревьями

В классе *BINARY_TREE* пока определены только три компонента, все они являются запросами: *item*, *left* и *right*. Мы можем добавить процедуру создания:

```
make (x: G)
        — Инициализация item значением x.
    do
        item := x
    ensure
        set: item = x
    end
Добавим в класс команды, позволяющие изменять поддеревья, и значение в корне:
add_left (x: G)
        — Создать левого сына со значением x..
    require
        no_left_child_behind: left = Void
    do
        create left.make (x)
    end
add_right … Аналогично add_left…
replace (x: G)
```

```
        -- Установить значение корня равным x.
    do item := x end
```

На практике удобно специфицировать *replace* как команду-присваиватель для соответствующего запроса, изменив объявление запроса следующим образом:

```
item: G assign replace
```

Это позволяет писать *bt.item*:= *x* вместо *bt.item.replace*(*x*).

Обходы бинарного дерева

Нет ничего удивительного в том, что, благодаря рекурсивной определенности, с бинарным деревом связываются многие рекурсивные методы. Функция *height* является одним из таких примеров. Приведем сейчас и другие. Предположим, что требуется напечатать все значения элементов, связанных с узлами дерева. Следующая процедура, добавленная в класс, выполняет эту работу:

```
print_all
        -- Печать значений всех узлов.
    do
        if left /= Void then print_all (left)end
        print (item)
        if right /= Void then print_all (right) end
    end
```

Здесь используется процедура *print* (доступная всем класса через общего родителя *ANY*), которая печатает подходящее представление значения типа *G* – родового параметра в классе *BINARY_TREE*[*G*].

Заметьте, структура *print_all* идентична структуре *hanoi*.

Хотя роль процедуры *print_all* состоит в печати всех значений, ее алгоритмическая схема не зависит от специфики операции, выполняемой над узлами дерева (*print* в данном случае). Процедура является примером **обхода** бинарного дерева: алгоритма, выполняющего однократно некоторую операцию над каждым элементом структуры данных в некотором заранее предписанном порядке обхода узлов. Обход является вариантом *итерации*.

Для двоичных деревьев наиболее часто используются три порядка обхода дерева, которые иногда называют соответственно инфиксным, префиксным и постфиксным порядками обхода.

Порядки обхода бинарного дерева

- **Inorder:** обход левого поддерева, посетить корень, обход правого поддерева.
- **Preorder:** посетить корень, обход левого поддерева, обход правого поддерева.
- **Postorder:** обход левого поддерева, обход правого поддерева, посетить корень.

В этих определениях «*посетить*» означает выполнение операции над отдельным узлом, такой как *print* в процедуре *print_all*; «обход» означает либо рекурсивное применение алгоритма для поддерева, либо отсутствие каких-либо действий, если поддерево пусто.

Префиксный порядок обхода, так же как и другие способы обхода, идущие, насколько это возможно, в глубину поддерева, называются также обходами «*вначале в глубину*», например, «*самый левый в глубину*».

Процедура *print_all* является иллюстрацией инфиксного способа обхода дерева. Достаточно просто записываются и другие способы обхода, например, для постфиксного обхода тело процедуры *post* имеет вид:

```
if left /= Void then post (left) end
if right /= Void then post (right) end
visit (item)
```

Здесь *visit* является операцией над узлом, такой как *print*.

> В запросах, предназначенных для повторного использования, нежелательно применять различные методы для вариантов данной схемы обхода, просто по причине изменения операции *visit*. Чтобы избежать этого, можно рассматривать операцию как аргумент метода обхода. Это возможно сделать в Eiffel, используя механизм **агентов**, описанный в последующих главах.

В качестве еще одной иллюстрации инфиксного обхода рассмотрим снова бинарное дерево выполнения *hanoi* для $n = 3$, где узлы уровня 0 опущены, поскольку не представляют интересной информации в данном случае.

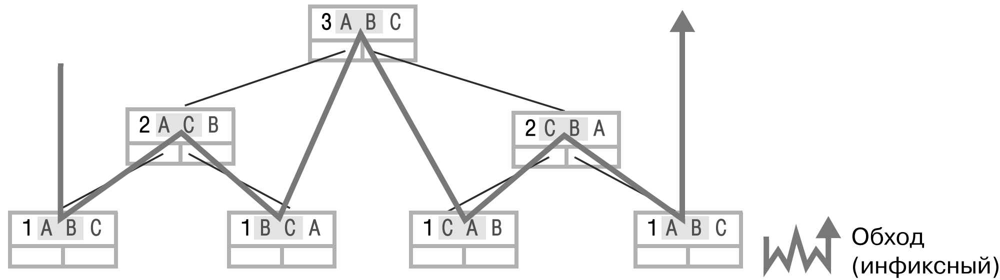


Рис. 14.14. Выполнение Hanoi в инфиксном порядке обхода

Процедура *hanoi* является матерью всех инфиксных обходов: обход левого дерева, если оно есть, посещение корня, обход правого поддерева, если оно есть. При посещении каждого узла выполняется операция *move*(*source, target*). Инфиксный порядок дает нужную последовательность ходов: **A B, A C, B C, A B, C A, C B, A B.**

Бинарные деревья поиска

Для произвольного бинарного дерева процедура *print_all*, реализующая инфиксный порядок, печатает значения узлов в произвольном порядке. Если же порядок важен для нас, то необходимо переходить к бинарным деревьям *поиска*.

Множество *G*, над которым определено общее бинарное дерево, может быть любым множеством. Для бинарных деревьев поиска предполагается, что на множестве *G* задано полное отношение порядка, позволяющее сравнивать любые два элемента *G*, задавая булевское вы-

ражение $a < b$, такое, что истинно одно из трех отношений: $a < b, b < a, a \sim b$. Примерами таких множеств могут служить *INTEGER* и *REAL* с отношением порядка <, но *G* может быть любым вполне упорядоченным множеством.

Как обычно, мы пишем $a <= b$, когда либо $a < b$, либо $a \sim b$. Аналогично, $b > a$, если $a < b$. Над вполне упорядоченным множеством можно определить бинарное дерево поиска.

Определение: бинарное дерево поиска

Бинарным деревом поиска над вполне упорядоченным множеством G называется бинарное дерево, такое, что для любого поддерева с корнем root со значением *item*, равным *r*, выполняются условия:

- если у корня root есть левое поддерево, то для любого его узла со значением *item*, равным *le*, справедливо: *le* < *r*;
- если у корня root есть правое поддерево, то для любого его узла со значением *item*, равным *ri,* справедливо: *ri* > *r*.

Значения в узлах левого поддерева меньше значения в корне, а в правом поддереве – больше. Это свойство применимо не только ко всему дереву в целом, но, рекурсивно, к любому поддереву, прямому или непрямому потомку корня. Это свойство будем называть **инвариантом бинарного дерева поиска**.

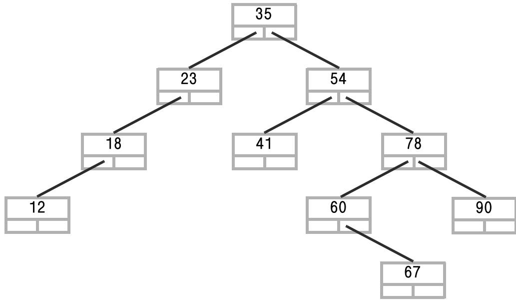


Рис. 14.15. Упражнение 14-У.3

Из этого определения следует, что все значения в узлах дерева должны быть различны. Мы для простоты будем использовать такое соглашение. Вполне возможно разрешать появление узлов с одинаковыми значениями; тогда отношение < и > в наших определениях пришлось бы заменить на <= и >=. В одном из упражнений придется адаптировать рассматриваемые алгоритмы на такой случай деревьев поиска.

Дерево, показанное на рисунке, является бинарным деревом поиска.

Процедура *print_all*, примененная к этому дереву, напечатает значения, упорядоченные по возрастанию, начиная с наименьшего значения 12.

Время программирования!

Печать упорядоченных значений

Используя приведенные на данный момент процедуры, постройте дерево, показанное на рисунке, затем напечатайте значения в узлах, вызвав *print_all*. Убедитесь, что значения упорядочены.

Производительность

Давайте рассмотрим причины, по которым деревья поиска являются полезными, как контейнерные структуры – потенциальные соперники хэш-таблиц. В самом деле, они обычно обеспечивают лучшую производительность, чем последовательные списки. В предположении случайного характера появления данных последовательный список, содержащий n элементов, обеспечивает:

- O(1) вставку (если элементы сохраняются в порядке вставки);
- O(n) для поиска.

Для бинарного дерева поиска обе операции имеют сложность O(log n), что намного лучше, чем O(n) для больших n (вспомните, что для нотации «О-большое» не имеет значения основание алгоритма). Проанализируем эффективность работы **полного** бинарного дерева, которое определяется тем, что для любого его узла оба поддерева, выходящие из узла, имеют одну и ту же высоту h:

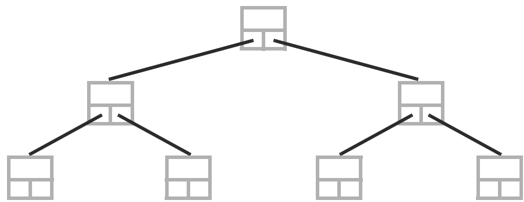


Рис. 14.16. Полное бинарное дерево

Нетрудно видеть, при помощи индукции по высоте h, что число узлов n в полном дереве высоты h равно $2^h - 1$. Как следствие, $h = \log_2(n+1)$. В полном дереве как поиск, так и вставка, используя приведенные ниже алгоритмы, выполняются за время O(log n), начиная работу в корне и следуя нисходящим путем к листьям дерева. В этом и состоит главная привлекательность бинарных деревьев поиска.

Конечно, большинство практических бинарных деревьев не являются полными. Если не повезет при вставке, то производительность может быть столь же плоха, как и для последовательного списка – O(n). К этому добавляются потери памяти, связанные с необходимостью хранения двух ссылок для каждого узла, в то время как для списка достаточно одной ссылки. На следующем рисунке показаны варианты «плохих» деревьев поиска:

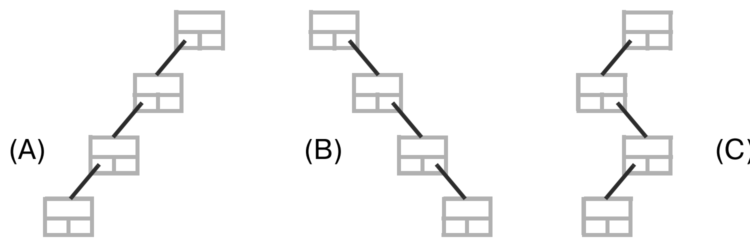


Рис. 14.17. Схемы бинарных деревьев поиска, являющиеся причиной поведения O(N)

При случайном порядке вставки бинарные деревья поиска остаются достаточно близкими к полным деревьям с поведением близким к O(log n). Можно гарантировать выполнение операций поиска, вставки и удаления за время O(log n), если пользоваться специальными вариантами таких деревьев – АВЛ-деревьями или черно-красными деревьями, которые являются почти полными деревьями.

Вставка, поиск, удаление

Приведем рекурсивную функцию поиска в бинарном дереве поиска (эта и следующая функция добавлены в класс, реализующий бинарные деревья поиска):

```
has (x: G): BOOLEAN
      -- Есть ли x в каком-либо узле?
   require
      argument_exists: x /= Void
   do
      if x ~ item then
         Result := True
      elseif x < item then
         Result := (left /=Void) and then left.has (x)
      else -- x > item
         Result := (right /= Void) and then right.has (x)
      end
   end
```

Алгоритм имеет сложность O(h), где h – высота дерева, что означает O(log n) для полного или почти полного дерева.

В этом варианте приводится простая нерекурсивная версия алгоритма:

```
has1 (x: G): BOOLEAN
      -- Есть ли x в каком-либо узле?
   require
      argument_exists: x /= Void
   local
      node: BINARY_TREE [G]
   do
      from
         node := Current
      until
         Result or node = Void
      invariant
   -- x не находится в вышерасположенных узлах на нисходящем пути от корня
      loop
         if x < item then
            node := left
         elseif x > item then
            node := right
         else
            Result := True
```

```
            end
        variant
            – (Высота дерева) - (Длина пути от корня к узлу)
        end
    end
```

Для вставки элемента можно использовать следующую рекурсивную процедуру:

```
put (x: G)
        – Вставка x, если он не присутствует в дереве.
    require
        argument_exists: x /= Void
    do
        if x < item then
            if left = Void then
                add_left (x)
            else
                left.put (x)
            end
        elseif x > item then
            if right = Void then
                add_right (x)
            else
                right.put (x)
            end
        end
    end
```

Отсутствие ветви **else** во внешнем **if** отражает наше решение не размещать дублирующую информацию. Как следствие, вызов *put* с уже присутствующим значением не имеет эффекта. Это корректное поведение («не жучок, а свойство»), о чем и уведомляет заголовочный комментарий. Некоторые пользователи могут, однако, предпочитать другой API с предусловием, устанавливающим **not** *has*(*x*).

Нерекурсивная версия остается в качестве упражнения.

Возникает естественный вопрос: как написать процедуру удаления – *remove*(*x*:*G*)? Это не так просто, поскольку нельзя просто удалить узел, содержащий *x*, если только это не лист дерева. Удаление листа сводится к обнулению одной из ссылок – *left* или *right*. Удаление произвольного узла приводило бы к нарушению инварианта бинарного дерева поиска.

Мы могли бы дополнять узел специальным булевским атрибутом, указывающим, удален ли узел фактически или нет. Но это решение слишком сложное, требует дополнительной памяти и мешает реализации других алгоритмов.

Что следует сделать, так это реорганизовать дерево, перемещая узлы, лежащие в основании узла с найденным значением *x*, так, чтобы сохранить истинность инварианта.

В удаляемый узел нужно поместить элемент дерева, лежащий ниже удаляемого элемента. Есть два кандидата на эту роль, позволяющие сохранить истинность инварианта:

- в левом поддереве – это элемент с максимальным значением (такой элемент является листом дерева или имеет только одну связь);
- в правом поддереве – это элемент с минимальным значением.

Если перемещаемый элемент является листом дерева, то после перемещения соответствующий узел удаляется. Если элемент имеет одну ссылку, то ссылка родителя перемещаемого элемента связывается с потомком перемещаемого элемента, тем самым перемещаемый узел исключается из дерева. Инвариант при этих операциях сохраняется.

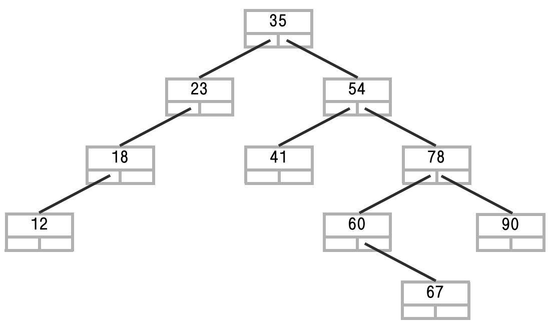


Рис. 14.18. Ранее построенное дерево

Предположим, что удаляется корень дерева – *remove* (35). Тогда в корень можно поместить либо элемент 23 (из левого поддерева), либо элемент 41 из правого поддерева.

Подобно поиску и вставке процесс удаления должен выполняться за время O(h), где h – это высота дерева. Процедура удаления остается в качестве упражнения.

Время программирования!

Удаление в бинарном дереве поиска

Напишите процедуру *remove*(x:G), которая удаляет элемент x, если он присутствует в дереве, сохраняя инвариант.

14.5. Перебор с возвратами и альфа-бета

Прежде чем исследовать теоретические основы рекурсивного программирования, полезно познакомиться еще с одним приложением, а точнее – с целым классом приложений, для которого рекурсия является естественным инструментом: алгоритмами перебора с возвратом. В имени отражена основная идея алгоритма: отыскивается решение некоторой проблемы, последовательно перебираются возможные пути; всякий раз, когда решение на данном пути заходит в тупик, происходит возврат назад к предыдущей развилке, из которой не все возможные пути были исследованы. Процесс заканчивается успешно, если на одном из путей достигается решение задачи. Неуспех в решении возникает тогда, когда ни на одном пути решение не получено или достигнут лимит времени, отведенный на поиск решения.

Перебор с возвратами применим к задаче, если каждое ее потенциальное решение может рассматриваться как последовательность выборов.

Бедственное положение застенчивого туриста

При необходимости достижения некоторой цели путешествия в незнакомом городе как к последней надежде можно прибегнуть к перебору с возвратом. Скажем, Вы находитесь в точке **A** (главная станция Цюриха) и хотите добраться до точки **B** (между главными зданиями ETH и Университета Цюриха):

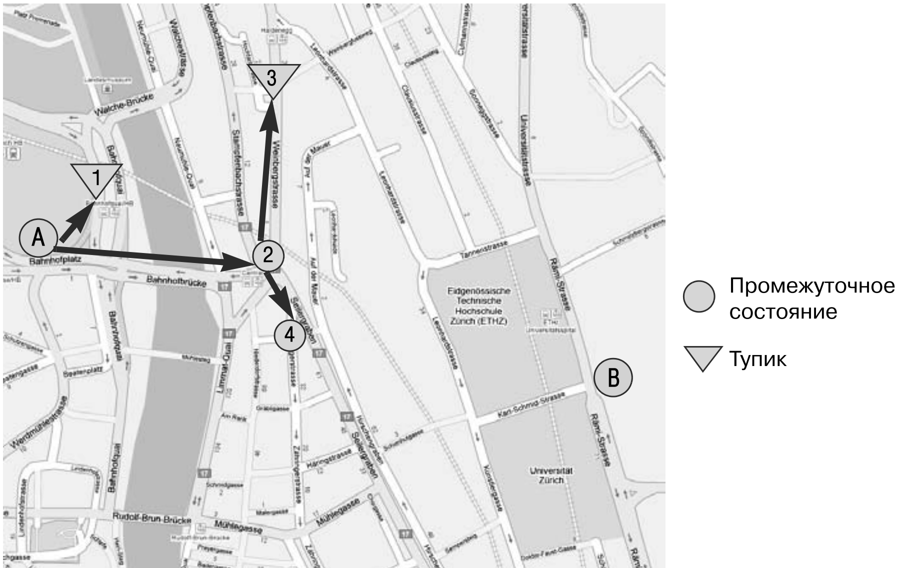


Рис. 14.19. Испытания и перебор с возвратами

Не имея карты и по застенчивости не отваживаясь спросить дорогу к цели, вы решили исследовать прилежащие улицы и проверять после каждой попытки, не достигнута ли цель (предполагается, что у вас есть фотография конечной точки). Вы знаете, что цель расположена на востоке, так что во избежание зацикливания все западно-ориентированные сегменты игнорируются.

На каждом этапе вы проверяете отрезки улиц, начиная с севера и двигаясь по часовой стрелке. Первая попытка приводит в точку 1. Здесь вы понимаете, что это не отвечает вашей цели, так как далее все возможные пути ведут на запад, то есть это тупик и нужно возвращаться назад в исходную точку **A.** Далее вы испытываете следующий выбор, приводящий в 2. Здесь есть несколько возможностей, и вначале вы проверяете путь, ведущий в 3, который оказывается тупиком, так что приходится вернуться в точку 2.

Если все «правильные» (не ведущие на запад) пути в 2 были исследованы и приводили к тупикам, то из 2 пришлось бы вернуться в точку **A**. Но в данной ситуации из 2 есть еще не исследованный выбор, ведущий в точку 4.

Процесс продолжается аналогичным образом, вы можете дополнить его самостоятельно, используя приведенную карту. Возможно, это не лучшая стратегия для путешествия в незнакомом городе, но иногда она может быть единственно возможной. Более важно, что она представляет общую схему метода «*проб и ошибок*», характерную для программирования перебора с возвратами. Схема может быть описана с помощью рекурсивной процедуры:

```
find ( p: PAT H): PATH
        - Решение, если оно существует, начинающееся в P.
    require
        meaningful: p /= Void
    local
        c: LIST [CHOICE]
    do
        if p.is_solution then
            Result := p
        else
            c := p.choices
            from c.start until
                (Result /= Void) or c.after
            loop
                Result := find ( p + c)
                c.forth
            end
        end
    end
```

Здесь применяются следующие соглашения. Выборы на каждом шаге описываются типом *CHOICE* (во многих ситуациях для этой цели можно использовать тип *INTEGER*). Есть также тип *PATH*, но каждый путь *path* – это просто последовательность выборов, и *p* + *c* – это путь, полученный присоединением *c* к *p*. Мы идентифицируем решение с путем, приводящим к цели, так что *find* возвращает *PATH*. По соглашению, если *find* не находит решения, то возвращается void. Запрос *is_solution* позволяет выяснить, найдено ли решение. Список выборов – *p.choices*, доступных из *p*, является пустым списком, если *p* – это тупик.

Для получения решения достаточно использовать *find*(*p0*), где *p0* – начальный пустой путь.

Как обычно, **Result** инициализируется значением **Void.** Если в вызове *find*(*p*) ни один из рекурсивных вызовов на возможных расширениях *p* + *c* не вырабатывает решения (в частности, если таких расширений вообще нет, поскольку *p.choices* пусто), то цикл завершится со значением *c.after*, и тогда *find*(*p*) вернет значение **Void**. Если это был начальный вызов *find*(*p0*), то процесс завершается, не достигнув положительного результата, в противном случае рекурсивно включается следующий вызов, пока не будет получен результат или не будут перебраны все возможности.

Если один из вызовов находит, что *p* является решением, то *p* возвращается в качестве результата, и, подымаясь вверх по цепочке вызовов, результат возвращается, поскольку **Result /= Void** является условием выхода.

Рекурсия дает четкую основу для управления такой схемой. Это естественный способ выразить природу метода проб и ошибок в алгоритмах перебора с возвратом – техническая реализация рекурсии заботится обо всех деталях. Для осознания этого вклада представьте на секунду, как пришлось бы программировать эту схему без рекурсии, – понадобилось бы сохранять всю предысторию пройденных путей (я не предполагаю, что вам необходимо написать полную не рекурсивную версию, по крайней мере, на данном этапе, когда еще не изучены способы реализации рекурсии, излагаемые позже в данной главе).

Дальнейшее обсуждение также показывает, как улучшить эффективность данного алгоритма, удаляя излишние записи. Например, фактически нет необходимости передавать путь как явно заданный аргумент, освобождая пространство в стеке вызовов. Вместо этого *p* может быть атрибутом, если мы добавим *p:= p+x* перед рекурсивным вызовом и *p:= p.head* после него (где *head* вырабатывает копию последовательности с удаленным последним элементом). Мы разработаем общий каркас, позволяющий безопасно выполнять такую оптимизацию.

Правильная организация перебора с возвратом

Общая схема перебора с возвратом требует некоторой настройки для практического использования. Прежде всего, в том виде, как она приведена, не гарантируется завершаемость. Чтобы убедиться в завершаемости любого выполнения, необходимо одно из двух:

- иметь гарантию (приходящую от проблемной области), что нет бесконечных путей, другими словами, что при расширении любого пути настанет момент, когда появится пустой список *выборов*;
- определить длину максимального пути и адаптировать алгоритм так, чтобы достижение границы рассматривалось как тупик. Вместо длины пути можно аналогично вводить ограничение по времени. Любой вариант реализуется простыми изменениями предыдущего алгоритма.

Дополнительно практическая реализация обычно может обнаруживать эквивалентность некоторых путей, например, для ситуации, представленной на рисунке:

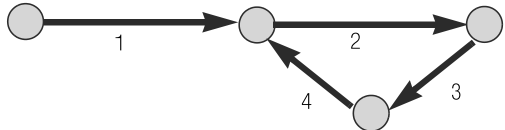


Рис. 14.20. Путь с циклом

Пути [1, 2, 3, 4], [1, 2, 3, 4, 2, 3, 4], [1, 2, 3, 4, 2, 3, 4, 2, 3, 4] и так далее являются эквивалентными. Пример, демонстрирующий нахождение маршрута без циклов, состоял в важной рекомендации – «никогда не уезжайте на запад, молодой человек». Конечно, этот совет не может служить общей рекомендацией и связан лишь с конкретной проблемной ситуацией. Чтобы справиться с циклом, необходимо сохранять информацию о ранее посещенных точках и игнорировать любой путь, ведущий к такой точке.

Перебор с возвратами и деревья

Для любой задачи, в решении которой может помочь перебор с возвратами, может помочь и построение модели в виде дерева. При установке соответствия будем использовать деревья, где каждый узел будет иметь произвольное число потомков, обобщая концепции, рассмотренные при определении бинарных деревьев. Путь в дереве (последовательность узлов) соответствует пути в алгоритме перебора (последовательность выборов). Дерево в примере с маршрутами представлено на рисунке 14.21.

Мы можем представлять всю карту города подобным образом: узлы отражают местоположение указанных на карте точек, дуги представляют отрезки улиц, соединяющих точки. В результате получается граф. Граф может быть деревом только в том случае, если в нем нет циклов. Если граф не является деревом, то на его основе можно построить дерево, называе-

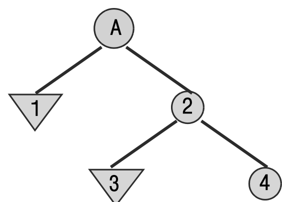


Рис. 14.21. Дерево путей при переборе с возвратами

мое остовным деревом графа, которое содержит все узлы графа и некоторые из его дуг. Для этого используются методы, упомянутые ранее, а также существует соглашение, позволяющее избежать цикла (никогда не ходите на запад) или построения путей из корня и исключающее любую дугу, которая ведет к ранее встречающемуся узлу. Вышеприведенное дерево является остовным деревом для части нашего примера, включающего узлы **A**, 1, 2, 3 и 4.

Для такого представления нашей задачи в виде дерева:

- решением является узел, удовлетворяющий заданному критерию (свойство, ранее названное *is_solution*, адаптированное для применения к узлам);
- выполнение алгоритма сводится к префиксному обходу дерева (самый левый в глубину).

В нашем примере этот порядок приведет к посещению узлов **A**, 1, 2, 3, 4.

Это соответствие указывает, что «Префиксный обход» и «Перебор с возвратами» основаны на одной и той же идее: всякий раз, когда мы рассматриваем возможный путь, мы разматываем все его возможные продолжения – все поддеревья узла, – прежде чем обращаемся к любому альтернативному выбору на уровне, представляющем непосредственных потомков узла. Например, если **A** на предыдущем рисунке будет иметь третьего потомка, то обход не будет его рассматривать, прежде чем не обойдет все поддеревья 2.

Единственное свойство, отличающее алгоритм перебора с возвратами от префиксного обхода, состоит в том, что он останавливается сразу же, как только найден узел, удовлетворяющий выбранному критерию.

Префиксный порядок был определен для бинарных деревьев следующим образом: посетить корень, затем обойти левое поддерево, затем правое. Порядок слева направо обобщается на произвольные деревья в предположении, что дети каждого узла упорядочены, – это не играет здесь существенной роли. Точно так же можно считать, что выборы для алгоритма перебора на каждом шаге упорядочены и просматриваются в соответствии с заданным порядком.

Минимакс

Интересным примером стратегии перебора, также естественным образом моделируемой деревом, является минимаксная техника, применяемая в играх, таких как шахматы. Она применима, если справедливы следующие предположения:

- в игре два игрока. Будем называть их Минни и Макс;
- для оценки ситуации, сложившейся в игре, будем использовать *оценочную функцию* с числовыми значениями, спроектированную так, что отрицательные значения хороши для Минни, а положительные – для Макса.

Примитивной оценочной функцией на примере игры в шашки может служить следующая функция, построенная в предположении, что Макс играет шашками черного цве-

та (black): (*mb* – *mw*) + 3 * (*kb* – *kw*), где *mb* и *mw* – число простых шашек соответственно у Макса и Минни, а *kb* и *kw* – число дамок, каждая из которых оценивается в три раза выше, чем простая шашка. Конечно, в хорошей игровой программе оценочная функция является более сложной.

Минни старается найти позицию, минимизирующую оценочную функцию, а Макс старается ее максимизировать.

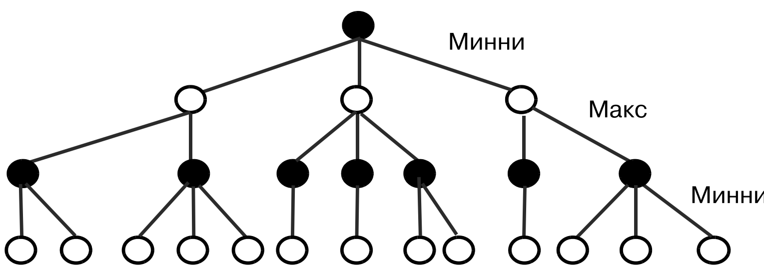


Рис. 14.22. Дерево игры

Каждый игрок использует минимаксную стратегию для выбора в текущей позиции одного из возможных ходов. Дерево моделирует игры такого вида, последовательные уровни дерева представляют ходы каждого из игроков.

На рисунке все начинается с позиции, где ход должна сделать Минни. Цель стратегии – позволить Минни среди ходов, доступных в данной позиции (три на рисунке), выбрать тот, который гарантирует лучший результат. В данном случае она старается получить минимальное значение оценочной функции на листьях дерева. Метод симметричен, так что Макс использует тот же механизм, стараясь максимизировать выигрыш.

Предположение о симметричности – основа для минимаксной стратегии, которая выполняет обход дерева, чтобы присвоить значение каждому узлу дерева.

M1 Значение листа является результатом применения оценочной функции к соответствующей позиции игры.

M2 Значение внутреннего узла, задающего ход Макса, является максимумом из значений детей этого узла.

M3 Значение внутреннего узла, задающего ход Минни, является минимумом из значений детей этого узла.

Значением игры в целом является значение, ассоциированное с корнем дерева. Для формирования стратегии следует в случаях M2 и M3 сохранять для каждого внутреннего узла не только значение, но и тот сыновний узел, обеспечивший оптимальную оценку. Вот иллюстрация стратегии, полученной в предположении, что значения в листьях вычислены с помощью некоторой оценочной функции:

Можно видеть, что значение в каждом узле является минимумом (для уровней 1 и 3) или максимумом (на уровне 2) значений сыновей узла. Оптимальный ход для Минни, обеспечивающий значение -7, состоит в выборе хода *C*.

Перебор с возвратами вполне подходит для минимаксной стратегии, в которой предварительно требуется вычислить значения в листьях и потом присваивать значения в узлах, подымаясь по дереву. Стратегия перебора с возвратами, основанная на принципе? «самый левый в глубину», отвечает такому подходу.

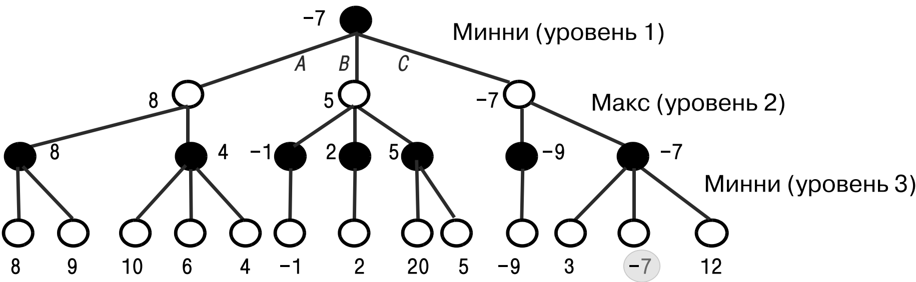


Рис. 14.23. Дерево игры с оценками

Следующий алгоритм, являясь вариацией ранее приведенной общей схемы перебора, реализует эту идею. Он представлен функцией *minimax*, возвращающей пару целых: value – гарантированное значение из начальной позиции *p*, и choice – начальный выбор, приводящий к этому значению. Аргумент *l* задает уровень, на котором появляется позиция *p* в ходе игры. Первый ход из этой позиции, возвращаемый как часть результата, является ходом Минни (как на рисунке), если *l* нечетно, и ходом Макса, если *l* четно.

```
minimax ( p: POSITION; l: INTEGER): TUPLE [value, choice: INTEGER]
   – Оптимальная стратегия(value + choice) на уровне l, начиная с позиции p.
   local
      next: TUPLE [value, choice: INTEGER]
   do
      if p.is_terminal (l ) then
         Result := [value: p.value; choice: 0]
      else
         c := p.choices
         from
            Result := worst (l )
            c.start
         until c.after loop
            next := minimax (p.moved (c.item), l + 1)
            Result := better (next, Result, l )
         end
      end
   end
```

Для представления результата используется кортеж, задающий значение и выбор.

Дополнительные функции *worst* и *better* дают возможность переключаться между Максом и Минни – игрок минимизирует на каждом нечетном ходе и максимизирует на четном.

```
worst (l: INTEGER): INTEGER
      – Худшее возможное значение для игрока на уровне l.
   do
      if l \\ 2 = 1 then Result := Max else Result := Min end
   end
```

```
   better (a, b: TUPLE [value, choice: INTEGER]; l: INTEGER):
                                                   TUPLE [value, choice: INTE-
GER]
         - Лучшее из a и b, в соответствии с их значениями для игрока на уровне l.
      do
         if l \\ 2 = 1 then
            Result := (a.value < b.value)
         else
            Result := (a.value > b.value)
         end
      end
```

Для определения худшего значения для каждого игрока вводятся две константы *Max* и *Min* (самое большое и самое малое значения).

Функция *minimax* предполагает существование следующих методов в классе *POSITION*:

- *is-terminal* указывает, что в данной позиции нет ходов, подлежащих исследованию;
- в этом случае *value* дает значение оценочной функции (запрос *value* может иметь предусловие *is-terminal*);
- для нетерминальной позиции пополняется список выборов *choices*, где каждый выбор представлен целым, задающим допустимый ход;
- если *i* является таким выбором, *moved*(*i*) дает позицию, полученную в результате применения соответствующего хода к текущей позиции.

Простейший способ убедиться, что алгоритм завершается, состоит в ограничении глубины перебора, задав *Limit* – предельное число уровней. Вот почему *is-terminal* в том виде, как она задана, включает уровень *l* как аргумент. Она может быть записана в виде:

```
   is_terminal (l: INTEGER): BOOLEAN
         – Следует ли исследовать уровень l или остановиться на текущей позиции?
      do
         Result := (l = Limit) or choices.is_empty
      end
```

На практике используются более сложные критерии остановки, например, алгоритм может сохранять затраты процессорного времени и останавливаться, когда достигнуто ограничение по времени.

Для выполнения мы вызываем *minimax*(*initial*, *l*), где *initial* задает начальную позицию игры. Уровень 1указывает на то, что ход принадлежит Минни.

Альфа-бета

Минимаксная стратегия всегда выполняет полный обход дерева допустимых ходов игры. Можно существенно улучшить эффективность работы, применяя оптимизацию, известную как «альфа-бета-стратегия», которая позволяет пропускать в процессе обхода целые поддеревья, «бесполезные» для достижения цели игры. Это прекрасная идея, заслуживающая внимания не только как «умное» решение, но и как пример усовершенствования рекурсивного алгоритма.

Альфа-бета имеет смысл только в предположениях, сделанных для минимаксной стратегии, когда стратегия одного игрока противоположна стратегии другого (один минимизирует, другой максимизирует), но во всем остальном стратегии игроков идентичны.

Идея пропуска поддерева основана на том, что игрок на уровне $l+1$ обнаруживает, что нет необходимости продолжать анализировать поддерево, поскольку он может получить на нем лучший результат, чем уже полученный. Но для его противника этот результат будет худшим, чем тот, что уже гарантирован ему на уровне l, и потому противник никогда не выберет это поддерево (напомним, что игроки всегда выбирают оптимальный ход с точки зрения принятой стратегии).

Предыдущий пример позволяет проиллюстрировать идею альфа-бета-стратегии. Рассмотрим ситуацию в процессе работы минимаксного алгоритма, когда исследовано несколько начальных узлов.

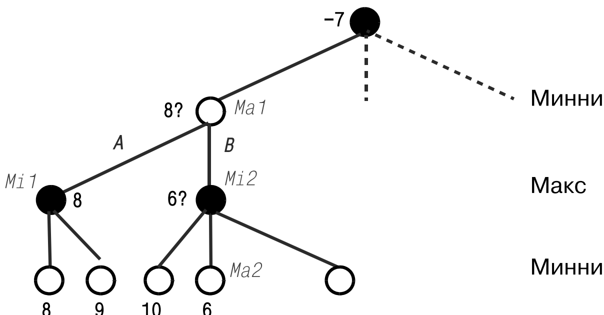


Рис. 14.24. Пропуск полезных поддеревьев

Мы находимся в процессе вычисления значения (максимума) для узла *Ma1* и, как часть этой цели, вычисления значения (минимум) для узла *Mi2*. Анализ первого поддерева для *Ma1* с корнем в *Mi1* позволил определить частичный максимум для *Ma1*, равный 8 (на рисунке он отмечен знаком вопроса, сигнализирующим, что вычисления еще не завершены). Это означает для Макса, что 8 он себе обеспечил и на меньшее не согласится. При анализе поддерева с корнем *Mi2* мы пришли к поддереву *Ma2* (к листу в данном конкретном случае, но вывод применим к любому поддереву), где значение равно 6, так что в любом случае Минни не выберет в *Mi2* значение, большее 6, а тогда ясно, что дальнейший анализ бесполезен, поскольку не окажет влияния на значение в вышестоящем узле *Ma1*. Так что, как только найдено значение 6 для *Ma2*, альфа-бета-стратегия прекратит анализ поддерева с корнем в *Mi2*.

На рисунке остался неисследованным только один лист, правее *Ma2*, но, конечно, в общем случае правее могло быть много узлов, являющихся корнями больших поддеревьев.

Эта оптимизация интересна сама по себе, но она дает и хорошую возможность отточить наши навыки рекурсивного программирования. Попытайтесь сами построить соответствующий метод, не заглядывая в решение, которое приводится ниже.

Время программирования!

Добавление альфа-бета в минимаксный алгоритм

Адаптируйте минимаксный алгоритм, приведенный ранее, так, чтобы он использовал стратегию «альфа-бета» для отбраковки бесполезных поддеревьев.

Расширение алгоритма достаточно просто. Методу понадобится еще один аргумент, задающий значение (если оно существует), которое для противника гарантированно находится на уровне непосредственно выше. Вот минимаксная процедура с добавленной альфа-бета стратегией:

```
alpha_beta ( p: POSITION; l: INTEGER; guarantee: INTEGER):
                                                    TUPLE [value, choice:
INTEGER]
    — Оптимальная стратегия(value + choice) на уровне l, начиная с позиции p.
      — Нечетный уровень минимизирует, четный - максимизирует.
    local
        next: TUPLE [value, choice: INTEGER]
    do
        if p.is_terminal (l ) then
            Result := [value: p.value; choice: 0]
        else
            c := p.choices
            from
                Result := worst (l )
                c.start
            until c.after or better (guarantee, Result, l - 1) loop
                next := minimax ( p.moved (c.item), l + 1), Result)
                Result := better (next, Result, l )
            end
        end
    end
```

Каждый игрок теперь останавливает анализ своего выбора, если противник может получить гарантированно лучший результат.

> Так как *better* определена без предусловия, то это означает, что она может принимать и уровень 0, так что допустимо передавать ей *l* - 1. Мы могли бы передавать и *l* + 1, поскольку вариантом *better*(*guarantee, Result, l-1*) является *better*(*Result, guarantee, l*), благодаря симметрической природе стратегии.

Рекурсивный вызов передает как «guarantee» на следующий уровень лучший Result, полученный на текущем уровне. Как следствие, альфа-бета останавливает обход узлов детей, когда срабатывает новый переключатель выхода *better*(*guarantee*, **Result**, *l*-1). Такая ситуация никогда не встретится для первого сыновнего узла, поскольку **Result** инициализирован значением *worst*.

Минимакс и альфа-бета широко используются в переборных алгоритмах в различных приложениях, где пространство поиска велико. Стратегия «альфа-бета» показывает, как важно в таких ситуациях улучшить стратегию поиска, чтобы избежать анализа возможных, но бесполезных вариантов.

14.6. От циклов к рекурсии

Вернемся назад, к общим проблемам рекурсии.

Мы уже видели, что некоторые рекурсивные алгоритмы – числа Фибоначчи, вставка и поиск в бинарных деревьях – имеют циклические эквиваленты. Что можно сказать в общем случае?

Фактически, нетрудно заменить любой цикл рекурсией. Рассмотрим произвольный цикл, данный здесь без указания инвариантов и варианта (хотя позже мы познакомимся с рекурсивными двойниками):

```
from Init until Exit loop Body end
```

Мы можем заменить его на

```
Init
loop_equiv

Здесь введена процедура:

loop_equiv
        – Используется условие выхода Exit и тело цикла Body.
    do
        if not Exit then
            Body
            Loop_equiv
        end
    end
```

В **функциональных языках** (таких как Lisp, Scheme, Haskell, ML) рекурсивный стиль является предпочитаемым, даже если доступны циклы. Мы могли бы также использовать рекурсию с первых шагов нашего курса, рассмотрев, например, анимацию линии метро, перемещающую красную точку, как рекурсивную процедуру:

```
Line8.start
animate_rest (Line8)
Вот как могла бы выглядеть сама процедура:
animate_rest (line: LINE)
        – Анимация станций линии метро, начиная от текущей позиции курсора
    do
        if not line.after then
            show_spot (line.item.location)
            line.forth
            animate_rest (line)
        end
    end
```

(более полная версия должна восстанавливать текущую позицию курсора).

Рекурсивная версия выглядит элегантно, но нет особых причин в рамках нашего курса предпочитать ее форме с применением циклов, мы и в дальнейшем будем использовать циклы.

Заключение могло бы быть другим, если бы мы основывались на функциональном языке программирования, где рекурсия является частью стиля, характерного для такого языка.

Но даже чисто теоретически важно понимать, что циклы не являются обязательной принадлежностью языка программирования и могут быть заменены рекурсией. Хорошим примером является программа *paradox,* демонстрирующая неразрешимость проблемы остановки. Рекурсия позволяет задать более выразительную версию:

```
recursive_paradox
        -- Завершается, если и только если не завершается.
    do
        if terminates ("C:\your_project") then
                recursive_paradox
        end
    end
```

Знание того, что всякий цикл может быть заменен рекурсией, немедленно порождает вопрос, а верно ли обратное – можно ли рекурсивную процедуру заменить процедурой, использующей циклы?

С примерами замены мы уже встречались – те же числа Фибоначчи, *has* и *put* для бинарных деревьев. Другие рекурсивные процедуры – *hanoi*, *height*, *print_all* – не имели свободного от рекурсии эквивалента. Для понимания того, что точно может быть сделано, необходимо более глубоко познакомиться со свойствами и смыслом рекурсивных программ.

14.7. Понимание рекурсии

Приобретенный опыт построения рекурсивных программ позволяет нам более глубоко исследовать смысл рекурсивных определений.

Неправильные циклы?

Прежде всего, вернемся назад и зададим весьма невежливый вопрос: а не является ли рекурсия «голым королем»? Другими словами, стоит ли что-либо за рекурсивным определением? Примеры, особенно примеры рекурсивных программ, свидетельствуют в их пользу, но некоторая доля сомнений все же остается. Мы все же находимся в опасной близости к определениям, не имеющим смысла, – к каким-то неправильным циклам. Рекурсия позволяет определять понятие в терминах самого понятия. Но, когда говорится:

Информатика занимается изучением информатики

то это общее место, тавтология, ничего не определяющая, в отличие от тавтологий в логике, которые доказываются и, следовательно, представляют интерес. Можно попытаться улучшить определение, сказав:

Информатика занимается изучением программирования, структур данных, алгоритмов, приложений, теоретическими вопросами и другими областями информатики

В определение добавлены полезные элементы, но оно все еще не является удовлетворительным определением. Подобным образом могут оказаться бесполезными и рекурсивные программы, такие как эта:

```
p (x: INTEGER)
        — Что в этом хорошего?
    do p (x) end
```

Эта программа выполняется для любых целочисленных аргументов, не производя никакого результата, работая бесконечно долго.

> «Бесконечно долго» – это математическая иллюзия. Фактически это означает, что для типичной реализации рекурсии компилятором на реальном компьютере программа будет работать, пока не переполнится стек вызовов, что станет причиной аварийного завершения программы.

Как избежать такого очевидного ошибочного использования рекурсии? Если попытаться понять, почему рекурсивные определения, с которыми мы сталкивались, кажутся интуитивно имеющими смысл, то можно выделить три интересных свойства.

Почувствуйте методологию

Правильно построенное рекурсивное определение

Полезное рекурсивное определение должно удовлетворять следующим требованиям:

R1 должна присутствовать хотя бы одна нерекурсивная ветвь;

R2 каждый вызов рекурсивной ветви должен иметь контекст, отличающийся от контекста, в котором эта ветвь была вызвана;

R3 для каждой рекурсивной ветви изменение контекста (R2) приближает к одному из нерекурсивных случаев (R1).

Для рекурсивных программ изменение контекста (R2) может заключаться в том, что вызов использует различное значение аргумента, как в вызове *r*(*n* -1) в программе *r*(*n*:*INTEGER*). Этот вызов применим к различным целям – *x*.*r*(*n*), где *x* не является текущим объектом. Изменение контекста может также означать, что вызов встречается после того, как программа изменила по меньшей мере одно поле по меньшей мере одного объекта.

Все рекурсивные программы, рассмотренные нами ранее, удовлетворяют этим требованиям.

- Тело *Hanoi*(*n*, ...) включает условный оператор **if** *n* > 0 **then** ... **end**, где все рекурсивные вызовы сосредоточены в **then**-ветви оператора, но поскольку **else**-ветвь отсутствует, то эта «пустая» ветвь при *n* = 0 определяет нерекурсивный вариант (R1). Рекурсивные вызовы имеют форму *Hanoi*(*n - 1*, ...), изменяя первый аргумент и порядок других аргументов (R2). Замена *n* на *n – 1*приближает контекст к нерекурсивному случаю *n* = 0 (R3).
- Рекурсивный метод *has* для бинарных деревьев поиска имеет нерекурсивные варианты для *x* = *item*, для *x* < *item*, если нет левого поддерева, и для *x* > *item*, если нет правого поддерева (R1). Рекурсивные вызовы имеют другую цель – *left* или *right*, отличающуюся от текущего объекта (R2). Каждый такой вызов приближается к листьям дерева, где рекурсия заканчивается (R3). Все эти утверждения справедливы и для других методов, работающих с деревьями поиска, например, *height*.
- В методе *animate_rest* – рекурсивной версии обхода линии метро, – когда курсор находится в положении *after*, срабатывает нерекурсивная ветвь (R1), ничего не делающая.

Рекурсивные вызовы не изменяют аргумент, но в процессе работы вызывается метод *line.forth*, изменяющий состояние линии (R2); при этом курсор передвигается ближе к состоянию *after*, где рекурсия заканчивается (R3).

Для рекурсивных понятий, не связанных с программами, условия R1, R2, R3 также должны выполняться.

- Мини-грамматика, определяющая понятие «*Операторы*», имеет нерекурсивный вариант – «*Присваивание*»;
- Все наши рекурсивно определенные структуры данных, такие как *STOP*, являются рекурсивными благодаря ссылкам, которые могут иметь значение void. В связных структурах значения void служат в качестве терминаторов, завершающих структуру.
- В случае рекурсивных программ комбинирование трех вышеприведенных правил предполагает понятие варианта, подобное варианту цикла, гарантирующего завершение цикла.

Почувствуй методологию

Вариант в рекурсии

Каждая рекурсивная программа должна быть объявлена с ассоциированным с рекурсией вариантом, целочисленной величиной, связанной с каждым вызовом, такой, что:

- предусловие программы гарантирует неотрицательность варианта;
- если выполнение программы начинается со значения *v* для варианта, то значение варианта *v1* для любого рекурсивного вызова удовлетворяет условию 0 <= *v1* < *v*.

Вариант может включать аргументы рекурсивного метода, а также другие элементы окружения, такие как атрибуты текущего объекта или другие объекты. Давайте посмотрим на примеры.

- Для *Hanoi*(*n*, ...) вариантом является *n*.
- Для *has*, *height, print_all* и других рекурсивных методов, связанных с обходом бинарных деревьев, вариантом является *node_height* – наибольшая длина пути от текущего узла до одного из листьев дерева.
- Для *animate_rest* вариантом является, как и для соответствующего цикла, *Line8.count* – *Line8.index* +1.

Специального синтаксиса для вариантов рекурсивных методов нет, но мы будем использовать комментарий в следующей форме, показанной для процедуры *Hanoi*(*n*, ...):

```
        -- variant n
```

Интересные случаи рекурсии

Хорошо определенные правила кажутся настолько разумными, что мы можем думать, что они являются не только достаточными, но и необходимыми правилами, чтобы рекурсивное определение имело смысл. Это и в самом деле справедливо для первых двух правил.

R1 Если все ветви определения являются рекурсивными, то невозможно выработать какой-либо экземпляр, который не был бы уже известен. В случае рекурсивных программ такое определение приводит к бесконечным вычислениям, на практике аварийно заканчивающимся из-за переполнения памяти.

R2 Если рекурсивная ветвь применяется к оригинальному контексту, то она не может выработать экземпляр, который не был бы уже известен. Для рекурсивной программы (например, *p(x: T)* с ветвью, которая вызывает *p(x)* для того же *x*, что и в начальном вызове, и где ничего не менялось), это приводит опять-таки к зацикливанию вычислений. Если речь не идет о программах и вычислениях, то такая ветвь бесполезна.

Другая ситуация – с правилом R3, где правило требует существования варианта у рекурсии, такого как аргумент *n* у *Hanoi*. Некоторые рекурсивные программы, которые завершаются, нарушают это свойство. Приведу два примера. Они не имеют практических применений, но высвечивают общие свойства, которые следует знать.

Функция 91 Маккарти была спроектирована Джоном Маккарти, профессором университета в Стэнфорде, создателем языка Лисп (в котором рекурсия играет центральную роль) и одного из создателей направления, получившего название «Искусственный интеллект». Определим ее следующим образом:

```
mc_carthy (n: INTEGER): INTEGER
        – Функция 91 Маккарти.
    do
        if n > 100 then
            Result := n - 10
        else
            Result := mc_carthy (mc_carthy (n + 11))
        end
    end
```

Для целых *n*, больших100, она возвращает значение *n* – 10. Это понятно. Значительно менее понятно из-за двойного рекурсивного вызова, какое же значение вернет функция для *n*, меньших 100, в том числе и для отрицательных значений, и вообще – закончатся ли вычисления. Оказывается, что во всех случаях, когда *n* меньше 100, функция завершает работу и возвращает значение 91, из-за чего функция и получила такое имя. Но очевидного варианта здесь нет, и внутренний рекурсивный вызов использует значение, большее начального *n*.

Вот еще один пример знаменитой программы и знаменитой проблемы, не получившей решения до настоящего момента:

```
bizarre (n: INTEGER): INTEGER
            – Функция, всегда возвращающая 1 для n нечетного и большего 1.
    require
        positive: n >= 1
    do
        if n = 1 then
            Result := 1
        elseif even (n) then
            Result := bizarre (n // 2)
        else      – для нечетных n, больших 1
            Result := bizarre ((3*n + 1) // 2)
        end
    end
```

Здесь используются операция // – деление нацело, и булевское выражение *even*(*n*), истинное для четных *n*. Два вхождения этой операции дают точное значение, поскольку применяются к четным числам. Понятно, что если функция возвращает результат, то он может быть только 1, производимой единственной нерекурсивной ветвью. Но завершается ли эта программа для любых *n*? Ответ *кажется* очевидным: «да» (можно написать программу и проверить ее на возможном диапазоне чисел. Доказана ее завершаемость на очень больших числах, но **общего решения пока нет**). Явного варианта рекурсии здесь нет, и видно, что в одной из ветвей рекурсивного вызова аргумент (3*n +1)//2 больше, чем *n*.

Эти интересные примеры являются экзотикой, но необходимо принимать в расчет их существование для общего понимания рекурсии. Они означают, что существуют некоторые рекурсивные определения, которые не удовлетворяют, казалось бы, разумным методологическим правилам, обсужденным выше, и все же вырабатывают хорошо определенные результаты.

Заметьте, что такие примеры, завершающиеся для каждого возможного аргумента, должны иметь вариант, так как для любого выполнения число рекурсивных вызовов определено и конечно, и следовательно, определяется состоянием программы во время вычисления, так что оно является функцией состояния, которая и может служить вариантом. Беда в том, что эта функция неизвестна, мы не можем ее выразить, а раз так, то ее теоретическое существование мало что дает практике.

Уже отмечалось, что невозможно автоматически определить – с помощью компилятора или другого инструментария, – существует ли у программы вариант в рекурсии, свидетельствующий о завершаемости процесса. Тем более невозможно автоматически спроектировать такой вариант, поскольку это означало бы возможность решения проблемы остановки.

На практике мы такие примеры оставляем без внимания и ограничиваем себя рекурсивными определениями, которые обладают всеми тремя свойствами – R1, R2, R3. В частности, когда вы пишете рекурсивную программу, следует всегда, как в оставшихся примерах этой главы, явно задавать вариант в рекурсии.

Определения, не требующие творчества

Даже с хорошо определенными правилами и наличием варианта у рекурсии мы все же не можем спать спокойно, используя рекурсию. Проблема в том, что рекурсивное определение не является определением в обычном смысле, поскольку оно может быть **креативным** – творческим.

Аксиома в математике является креативной: она говорит нам нечто такое, что не может быть выведено из известных уже фактов. Примером является аксиома о целых в математике, которая говорит, что для числа *n'*, следующего за *n*, справедливо *n* < *n'*. Фундаментальные законы в естественных науках также креативны, например, закон, утверждающий о невозможности двигаться со скоростью, превышающей скорость света.

Теоремы в математике, так же как и аналогичные результаты в физике, с рассматриваемой точки зрения «творческими» не являются, поскольку могут быть выведены из аксиом и известных уже законов. Они интересны сами по себе и позволяют нам более просто выводить новые утверждения, но они не добавляют новых предположений, являясь следствиями уже сделанных ранее.

Определение, так же как и теорема, не должно быть творческим. Определение дает новое имя объекту нашего мира, но все утверждения, которые можно выразить с использованием

определения, могут быть сделаны и без помощи этого определения. Мы хотим более просто выражать утверждения, используя введенное определение, а иначе зачем бы понадобилось его вводить. Но мы знаем, что можно обойтись и без определения. Пусть сказано:

```
Определим x² для любого x, как x * x
```

Ничего нового такое определение в математику не добавляет: просто разрешается использовать новую нотацию для умножения. Любое свойство, которое может быть доказано с использование новой нотации, может быть доказано и без нее, по сути, заменой x^2 в соответствии с определением.

Символ ≜, который мы использовали для обозначения «определено как» (начиная с БНФ-продукционных правил грамматики) предполагает этот некреативный характер определения. Но давайте рассмотрим рекурсивное определение в форме:

```
f ≜ some_expression                                            [1]
```

Здесь *some_expression* включает *f*. Теперь наш принцип больше не выполняется! Всякая попытка заменить *f* в определении *some_expression* на *some_expression* не устраняет *f*, так что реально мы ничего не определили. До тех пор, пока мы не найдем удобного, некреативного смысла для определений, подобных **[1],** мы должны быть терминологически аккуратны. По этой причине символ ≜ будет использоваться только для нерекурсивных определений, а такое свойство, как **[1],** будет задаваться равенством

```
f = some_expression                                            [2]
```

Это равенство можно рассматривать как уравнение, решением которого выступает *f*. Говоря о рекурсивных «определениях», для корректности будем заключать второе слово в кавычки.

Взгляд на рекурсивные «определения» снизу вверх

Изолировав пока рекурсию и поместив ее в карантинную зону, полезно посмотреть на рекурсивные программы и рекурсивные «определения» в целом с позиции «снизу вверх». Я надеюсь, что это удалит легкое головокружение, которое остается, когда видишь определения программ, которые – частично – сами себя вызывают.

Рекурсивные «определения» пишутся «сверху вниз», определяя смысл понятия в терминах того же понятия для «меньшего» контекста – меньшего с точки зрения варианта рекурсии. Например, *Fibonacci* для *n* выражается через *Fibonacci* для $n - 1$ и $n - 2$.

Взгляд «снизу-вверх» предоставляет другую интерпретацию того же определения, трактуя это другим способом, как механизм, который создает новое значение на основе уже существующих. Начнем с рассмотрения функции. Для любой функции *f* можно построить граф этой функции как множество пар [*x*, *f*(*x*)] для каждого применимого *x*. Граф для функции *Fibonacci* задается множеством

```
F ≜ {[0, 0], [1, 1], [2, 1], [3, 2], [4, 3], [5, 5], [6, 8], [7, 13] …}
```

Он содержит все пары [*n*, *Fibonacci*(*n*)] для всех неотрицательных *n*. Этот граф содержит всю информацию о функции. Визуально этот граф можно представить в следующей форме:

Рис. 14.25. Граф функции (Фибоначчи)

В верхней строчке показаны возможные аргументы функции, в нижней – соответствующие значения функции.

Дать функции рекурсивное определение – это все равно, что сказать, что ее граф *F* – как множество пар – удовлетворяет некоторому свойству

```
F = h (F )                                                        [4]
```

Здесь *h* рассматривается как некоторая функция, применимая к такому множеству пар. Это подобно уравнению, которому *F* должна удовлетворять, и известному как уравнение **неподвижной точки**. Неподвижной точкой – решением такого уравнения – является некоторый математический объект, в данном случае функция, который остается инвариантом при некоторых трансформациях, в данном случае – задаваемых функцией *h*.

Определим рекурсивно функцию *Fibonacci*:

```
fib (0) = 0
fib (1) = 1
fib (i) = fib (i - 1) + fib (i - 2)     - Для i >1
```

Такое определение эквивалентно тому, что граф *F*, рассматриваемый как множество пар, удовлетворяет уравнению неподвижной точки **[4],** где *h* – это функция, которая, получив множество пар, вырабатывает новое множество, содержащее следующие пары:

G1		каждую пару, уже содержащуюся в *F*
G2	[0, 0]	— Пару для $n = 0$: [0, *fib*(0)]
G3	[1, 1]	— Пару для $n = 1$: [1, *fib*(1)]
G4		каждую пару в форме $[i, a + b)]$ для некоторого *i*, такого, что *F* содержит пары $[i\text{-}1, a]$ и $[i\text{-}2, b]$

Мы можем использовать эту точку зрения, чтобы дать рекурсивному «определению» точный смысл, свободный от всякой рекурсивной загадочности. Мы начинаем с графа F_0, который пуст (не содержит пар). Далее мы определяем

$$F_1 \triangleq h(F_0)$$

Оно имеет смысл и означает, что F_1 задается множеством {[0, 0], [1, 1]} – парами, определяемыми правилами G2 и G3. Правила G1 и G4 в данном случае неприменимы, так как F_0 пусто. Затем мы снова применяем *h*, чтобы получить

$$F_2 \triangleq h(F_1)$$

Здесь G2 и G3 нам не дают ничего нового, так как пары [0, 0] и [1, 1] уже присутствуют в F_1, но G4 создает новую пару из существующих – [2, 1]. Продолжая, мы определяем последовательность графов, начиная с F_0 и определяя $F_i = h(F_{i-1})$. Теперь рассмотрим *F* как бесконечное объединение:

$$\bigcup_{i \in N} F_i$$

Здесь N – это множество натуральных чисел. Достаточно просто видеть, что F удовлетворяет свойству **[4].**

Такова нерекурсивная интерпретация – семантика, – которую мы дали рекурсивному «определению» функции *Fibonacci*.

В общем случае уравнение неподвижной точки в форме **[4]** на графах функции, устанавливающее эквивалентность F и $h(F)$, представляет решение в виде графа функции

$$F \triangleq \bigcup_{i \in N} F_i$$

Здесь F_i – это последовательность графов функции:

```
F₀   ≜  { }          – Пустое множество пар
Fᵢ   ≜  h(Fᵢ₋₁)      – Для i > 0
```

Подход, основанный на неподвижной точке, является базисом интерпретации «снизу вверх» рекурсивных вычислений. Он частично удаляет загадочность из этих определений, поскольку рассматривает рекурсивное определение как уравнение неподвижной точки и допускает решение, полученное как результат объединения (подобно пределу последовательности в математическом анализе) последовательности графов функции.

> Отсюда непосредственно следует требование, что любое полезное рекурсивное «определение» должно иметь нерекурсивную ветвь. Если бы это было не так, то последовательность, начинающаяся с пустого множества пар $F_0 = \{\}$, никогда бы не создавала новых пар, поскольку во всех случаях определения h, подобно G1 и G4 для Фибоначчи, новые пары создаются из существующих, а их нет в пустом множестве.

Данный подход редуцирует рекурсивное «определение» к хорошо известному, традиционному понятию индуктивного определения последовательности.

Функция Фибоначчи является хорошим примером для понимания концепции, но, вероятно, она не поражает впечатления. Во всех учебниках по математике она определяется индуктивно как последовательность чисел. Это в компьютерном мире мы рассматриваем ее как рекурсивную функцию. Так что мы не узнали ничего нового о самой функции, а просто познакомились с разными точками зрения. Давайте посмотрим, можно ли научиться чему-нибудь на других примерах.

Интерпретация «снизу вверх» конструктивных определений

Понимая природу «снизу вверх», можно дать ясное понимание смысла рекурсивного определения понятия «тип». Как вы помните, тип определяется следующим образом:

T1 базисный класс, такой как *INTEGER* или *STATION*;

T2 родовое порождение в форме *C* [*T*], где *C* является универсальным классом и *T* – это тип.

Правило T1 определяет нерекурсивный случай. Взгляд «снизу вверх» позволяет нам понимать данное определение как построение множества типов в виде последовательности слоев. Ограничившись для простоты одним родовым параметром, имеем:

- слой L0 включает все типы, построенные из базисных классов: *INTEGER*, *STATION* и т.д.;
- слой L1 имеет все типы в форме *C* [*X*], где *C* универсальный класс, а *X* принадлежит уровню L0: *LIST* [*INTEGER*], *ARRAY* [*STATION*] и т.д.;

- в общем случае слой L*n* для любого $n > 0$ имеет все типы в форме $C [X]$, где X принадлежит уровню L*i* для $i < n$.

Таким образом, мы получаем все типы – базисные и полученные в результате родового порождения.

Башни, «снизу вверх»

Взглянем теперь «снизу вверх» на Ханойские башни. Программу можно рассматривать как рекурсивное определение последовательности ходов. Давайте обозначим такую последовательность как <**A** → **B**, **C** → **A**, ...>, означающую, что первым ходом переносится диск с вершины стержня **A** на **B**, затем с **C** на **A** и так далее. Пустая последовательность будет <>, а конкатенация последовательностей задается знаком «+», так что

<A → B, C → A> + <B → A> дает <A → B, C → A, B → A>.

Тогда мы можем выразить рекурсивное решение как функцию *han* с четырьмя аргументами (целое и три стержня), вырабатывающую последовательность ходов и удовлетворяющую уравнению неподвижной точки

```
han (n, s, t, o) =
    < >                                                    – Если n = 0    [5]
    han (n - 1, s, o, t) +<s → t> + han (n - 1, o, t, s)     – Если n > 0   [6]
```

Функция определена, если *n* положительно и значения *s*, *t*, *o* (сокращения для *source*, *target*, *other*) различны – мы используем их, как и ранее, для обозначения стержней '**A**', '**B'**, '**C**'.

Конструирование функции, решающей уравнение, просто: **[5]** позволяет инициализировать граф функции для $n = 0$ в следующей форме:

```
[(0, s, t, o), < > ]
```

Обозначим через H_0 эту первую часть графа, содержащую 6 пар, которые включают все возможные перестановки стержней. После этого можно использовать **[6]** для получения множества пар H_1, содержащего значения для $n = 1$, где пары имеют вид:

```
[(1, s, t, o), <s →t>]
```

Здесь учитывается, что конкатенация < > + *x* или *x* + < > дает *x*. Следующая итерация **[6]** даст нам H_2, чьи пары имеют вид:

```
[(2, s, t, o), f1 + <s →t> + g1]
```

Это верно для всех *s*, *t*, *o* таких, что H_1 содержит как пару [(1, *s*, *o*, *t*), *f1*], так и пару [(1, *o*, *t*, *s*), *g1*].

Последующее итерирование позволит построить граф H_3. Полный граф – разумеется, бесконечный, поскольку включает пары для всех возможных *n*, – задает множество всех пар во всех элементах последовательности:

$$\bigcup_{i \in N} H_i$$

Теперь я смею надеяться, что вы получили достаточное представление о подходе «снизу вверх» к рекурсивным вычислениям и сможете написать программу построения графа.

Время программирования!

Построение графа функции
Напишите программу (не используя рекурсии), создающую последовательно элементы множеств H_0, H_1, H_2 ... для *Hanoi*.

Связанное с этой задачей упражнение попросит вас определить (без программирования) математические свойства графа.

Еще одно важное упражнение потребует применить подобный анализ к обходу бинарного дерева. Вы должны будете спроектировать модель для представления решения, подобную той, что использована в данной задаче, – вместо последовательности ходов должна появиться последовательность узлов.

Грамматики как рекурсивно определенные функции

Подход «снизу вверх», в частности, применим и для рекурсивных грамматик, как в нашем небольшом примере:

```
Instruction ≜ ast | Conditional
Conditional ≜ ifc Instruction end
```

Здесь введены сокращения: **ifc** представляет «**if** Condition **then**» и **ast** представляет «Assignment», оба рассматриваются как терминалы в данном обсуждении.

Достаточно просто видеть, как генерировать последовательные предложения языка, интерпретируя создаваемые продукции в стиле неподвижной точки:

```
ast
ifc ast end
ifc ifc ast end end
ifc ifc ifc ast end end end
```

Генерация продукций может быть продолжена.

С этих же позиций может быть проанализировано предыдущее обсуждение небольшого языка Game.

Данный подход обобщается на произвольные грамматики, рассматривая матричное представление описания БНФ.

14.8. Контракты рекурсивных программ

Мы уже научились поставлять наши классы и их методы с контрактами, устанавливающими их корректность: предусловия и постусловия методов, инварианты класса. Те же подходы, применяемые к алгоритмам, приводят к заданию вариантов и инвариантов цикла. Как рекурсия вписывается в эту картину?

Мы уже познакомились с понятием **варианта для рекурсии**. Если метод рекурсивен непосредственно или косвенно, то следует включать вариант для рекурсии в его описание. Как отмечалось, специальный синтаксис для этих целей отсутствует, так что приходится добавлять отдельное предложение в заголовочный комментарий метода:

```
– variant: expression
```

Рекурсивный метод, подобно всякому методу, может иметь предусловие и постусловие. Поскольку ответственность за проверку предусловия всегда лежит на клиенте – вызывающей программе, а в данном случае клиентом является сам рекурсивный метод, то новинкой будет необходимость убедиться, что все рекурсивные вызовы внутри метода (для косвенной рекурсии – в ассоциированных методах) удовлетворяют предусловию.

Вот пример процедуры *Hanoi* с более полными контрактами, новыми предложениями, записанными в виде комментариев:

```
hanoi (n: INTEGER; source, target, other: CHARACTER)
       –Перенос n дисков из source на target,используя other
       –в соответствии с правилами игры Ханойская Башня
       – invariant: диски на каждом стержне образуют пирамиду,
       – следуя в порядке уменьшения размеров.
       – variant: n
    require
       non_negative: n >= 0
       different1: source /= target
       different2: target /= other
       different3: source /= other
       – source имеет n дисков; target и other пусты  - без дисков
    do
       if n > 0 then
          hanoi (n–1, source, other, target)
          move (source, target)
          hanoi (n–1, other, target, source)
       end
    ensure
       – Диски, ранее находившиеся на source, теперь перенесены на target,
       – сохраняя прежний порядок,
       – other находится в исходном состоянии.
    end
–invariant: текст комментария
```

Предлагаемая конструкция не является частью языка, но подчиняется следующим соглашениям.

- Если инвариант рекурсии является псевдокодом, заданным комментарием, как в данном примере, то он не повторяется в предусловии и постусловии, (здесь это означает, что в предусловии и постусловии ничего не говорится о требованиях сортировки дисков по размерам).
- Любое формальное предложение инварианта рекурсии (булевское выражение) должно включаться в предусловие и постусловие.

14.9. Реализация рекурсивных программ

Рекурсивное программирование хорошо работает в некоторых проблемных областях, что проиллюстрировано примерами этой главы. Когда рекурсия облегчает вашу работу, нужно применять ее, не раздумывая, так как в современных языках программирования рекурсия считается само собой разумеющейся.

Обычно на уровне машинного кода отсутствует прямая поддержка рекурсии. Компиляторы для языков высокого уровня должны отображать рекурсивно выраженный алгоритм в нерекурсивный. Применяемая при этом техника, несомненно, важна для разработчиков компиляторов, но и для вас, даже если вы и не собираетесь написать компилятор, полезно познакомиться с базисными идеями, как для лучшего понимания рекурсии, так и для осознания потенциальных проблем производительности, связанных с реализацией рекурсии.

Мы рассмотрим некоторые рекурсивные схемы и спросим себя, как, если язык не допускает рекурсию, можно было бы спроектировать нерекурсивную версию, называемую также итеративной, доставляющую те же результаты.

Рекурсивная схема

Рассмотрим рекурсивную процедуру *r*, содержащую собственный вызов:

```
r (x: T)
    do
        code_before
        r (y)
        code_after
    end
```

Здесь могло бы быть несколько рекурсивных вызовов, но мы пока рассматриваем только один. Что это означает, если вернуться к взгляду «сверху вниз»?

Наличие рекурсии влечет, что ни начало кода метода, ни его конец не являются тем, на что они претендуют (быть началом и концом).

- Когда выполняется *code_before*, то это вовсе не означает, что выполнение инициировано вызовом метода клиентом *a.r*(*y*) или неквалифицированным вызовом *r*.(*y*), – это может быть результатом работы экземпляра *r*, вызывающего себя рекурсивно.
- Когда *code_after* завершается, это вовсе не означает завершение истории *r*: это может быть просто завершение одного рекурсивно вызванного экземпляра. В этом случае следует подвести итоги выполнения последнего вызванного экземпляра *r* и продолжить выполнение предыдущего экземпляра.

Программы и экземпляры их выполнения

Ключевой новинкой последнего наблюдения является понятие **экземпляра** (называемого также **активацией**) программы. Мы знаем, что классы имеют экземпляры – «объекты», создаваемые при выполнении ОО-программы. Теперь подобным образом начнем рассматривать и методы класса.

В любой момент выполнения программы состояние вычислений характеризуется цепочкой вызовов, как показано на рисунке. Корневая процедура *p* вызывает *q*, которая, в свою очередь, вызывает *r*... Когда выполнение программы в цепочке завершается, скажем, *r*, от-

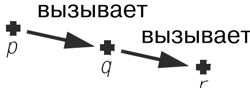


Рис. 14.26. Цепочка вызовов без рекурсии

ложенное выполнение вызывающей программы, здесь *q*, подводит итоги и продолжается с той точки, где оно было прервано при вызове *r*.

В отсутствие рекурсии не было особой необходимости явного введения понятия экземпляра метода, так как во время выполнения существовал только один активный экземпляр. При рекурсии цепочка вызовов может включать два или более экземпляров одного и того же метода. При прямой рекурсии они будут следовать друг за другом:

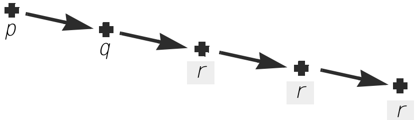


Рис. 14.27. Цепочка вызовов при прямой рекурсии

Например, вызов *hanoi*(2, *s*, *t*, *o*) непосредственно запустит вызов *hanoi*(1, *s*, *o, t*), который вызовет *hanoi*(0, *s*, *t*, *o*). В этом состоянии будем иметь три экземпляра процедуры в цепочке вызовов.

Подобная ситуация существует и при косвенной рекурсии:

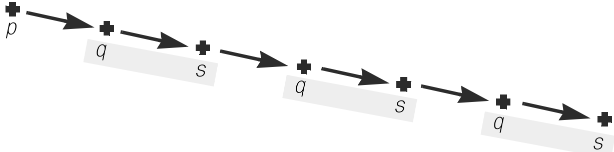


Рис. 14.28. Цепочка вызовов при косвенной рекурсии

Сохранение и восстановление контекста

Все экземпляры метода разделяют его код. Экземпляры различаются не кодом, а контекстом выполнения. Мы уже обращали внимание на то, что для правильно построенных рекурсивных методов при каждом вызове контекст экземпляра должен отличаться, по крайней мере, одним элементом от контекста других экземпляров. Контекст, характеризующий экземпляр, включает:

- значения фактических аргументов метода, если они есть, заданные для данного конкретного вызова;
- значения локальных переменных, если они есть;
- точку вызова в тексте вызывающего метода, позволяющую продолжить выполнение после завершения вызванного экземпляра.

При изучении того, как стеки поддерживают выполнение программ в современных языках программирования, мы познакомились со структурой данных, представляющей контекст выполнения метода и называемой **записью активации**.

При рекурсии каждой активации нужен собственный контекст. Так что остаются только две возможности реализации.

I1 Мы можем обратиться к *динамическому распределению*. Всякий раз, когда стартует очередной экземпляр рекурсивного метода, создается новая запись активации, содержащая контекст экземпляра. Она используется для доступа к фактическим аргументам и локальным переменным, она применяется и при завершении работы экземпляра, чтобы можно было продолжить работу вызывающей программы, которая продолжит работу с собственной записью активации.

I2 В целях экономии памяти можно заметить, что не всегда требуется создавать собственную запись активации, – как обычно, вместо сохранения данных можно перейти к их повторному вычислению. Такое возможно, если преобразование контекста **обратимо**, и разумно, если потери времени менее значимы, чем дополнительная память на хранение контекста. Рекурсивный вызов в процедуре *hanoi*(*n*, ...) имеет вид *hanoi*(*n-1*, ...). Вместо того, чтобы хранить n в активационной записи, сохранять значение *n* -1 в новой записи, можно, как при статическом распределении, в самом начале отвести память для хранения *n*. При вызове нового экземпляра значение *n уменьшается* на 1, а при завершении *увеличивается* на 1.

Два подхода не являются исключающими друг друга. Можно использовать подход I2 для элементов контекста, допускающих простую трансформацию, как с аргументом *n* в методе *hanoi*(*n*, ...), и создавать запись активации для остальных элементов контекста. Как всегда, решение принимается на основе компромисса «память или время».

Использование явного стека вызова

Из двух рассмотренных стратегий управления контекстом рассмотрим вначале первую, основанную на явных записях активации.

Подобно записям активации, динамически создаются и *объекты* в результате выполнения оператора **create**. Память программы, предназначенная для динамического распределения, называется **кучей** (heap). Но для записей активации нет необходимости использовать кучу, так как образцы активации и деактивации просты и предсказуемы.

- Вызов метода требует создания записи активации.
- По завершении работы метода становится активной запись вызвавшего метода, а о записи активации отработавшего метода можно забыть (она никогда больше не понадобится, так как новому вызову требуется собственная запись).

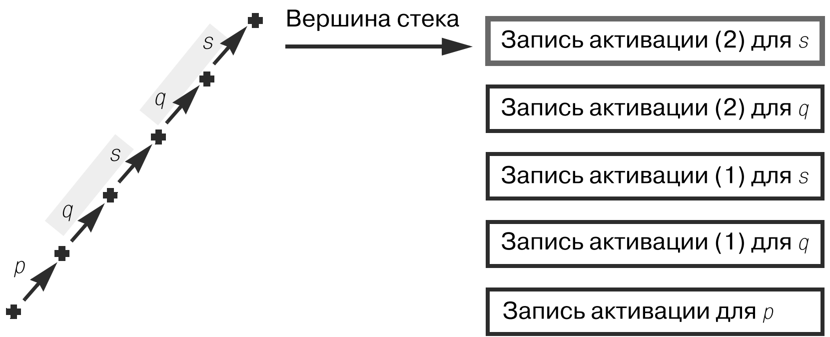


Рис. 14.29. Цепочка вызовов и соответствующий стек записей активации

Такая стратегия работы соответствует уже хорошо известной стратегии LIFO «последний пришел – первый ушел», для реализации которой применяется структура данных – стек. Стек записей активации задает цепочку вызовов, что показано на следующем рисунке.

Мы уже ранее встречались со стеком записей активации – он назывался стеком **вызовов**, сохраняющим историю вызовов методов во время выполнения. Если вы программируете на языке, поддерживающем рекурсию, то создание стека вызовов выполняется компилятором. Сейчас же мы посмотрим, как это можно сделать самому.

Вам может понадобиться явный стек записей активации, если по каким-либо причинам вы захотите написать итеративный вариант рекурсивного метода.

- Для получения доступа к локальным переменным и аргументам метода используйте соответствующие поля записи активации в вершине стека.
- Вместо рекурсивного вызова: создайте новую запись активации, инициализируйте ее значениями аргументов и положением точки вызова, поместите ее в стек и перейдите в начало кода, выполняющего метод.
- Вместо возврата: возвращайтесь, только если стек пуст (нет приостановленных вызовов, ждущих своей очереди); в противном случае восстановите значения аргументов и локальных переменных, удалите использованную запись активации из стека, перейдите к подходящему оператору кода метода в точку, прерванную вызовом только что завершенного метода.

Заметьте, обе трансляционные схемы вызова и возврата требуют оператора **goto** для перехода в нужную точку кода. Это прекрасно при работе с машинным кодом, но при работе с языком высокого уровня такого перехода стараются избежать, и в языке Eiffel оператора **goto** просто нет. В этом случае приходится временно написать имитацию **goto** и промоделировать его подходящими структурными средствами.

Основы исключения рекурсии

Давайте посмотрим, как эта схема работает для тела процедуры *hanoi* с ее двумя рекурсивными вызовами. Будем использовать стек записей активации, называемый просто *stack*:

```
stack: STACK [RECORD]
```

Вспомогательный класс *RECORD* задает запись активации:

```
note
    description: "Данные, связанные с экземпляром метода"
class RECORD create
    make
feature — Инициализация полей
    make (n: INTEGER; c: INTEGER; s, t, o: CHARACTER)
            — Инициализация полей записи: count, call, source, target, other.
        do
            count := n ; call := c; source := s ; target := t ; other := o
        end
feature — Access
    count: INTEGER.
            — Число дисков.
    call: INTEGER
            — Идентифицирует рекурсивный вызов: 1 - первый вызов, 2 - второй.
```

```
        source, target, other: CHARACTER
            -- Стержни.
    end
```

(вместо полноценного класса можно было бы использовать кортеж). Экземпляр класса представляет контекст вызова: число перемещаемых дисков (*count*), три стержня в порядке их использования в вызове, и *call*, задающее, первый или второй рекурсивный вызов будет применен.

```
    hanoi (n: INTEGER; source, target, other: CHARACTER)
        do
            if n > 0 then
                hanoi (n-1, source, other, target)       -- Первый вызов
                move (source, target)
                hanoi (n-1, other, target, source)   -- Второй вызов
            end
        end
```

Мы задействуем стек записей активации для реализации нерекурсивной версии процедуры, временно использующей **goto**:

```
    iterative_hanoi (n: INTEGER; source, target, other: CHARACTER)
        local -- Нам необходимы локальные переменные, представляющие аргументы
                -- последовательных вызовов:
            count: INTEGER
            x, y, z, t: CHARACTER
            call: INTEGER
            top: RECORD
        do    -- Инициализация локальных переменных:
            count := n; x := source; y := target; z := other
    start: if count > 0 then
                        -- Трансляция hanoi (n-1, source, other, target):
                stack.put (create {RECORD}. make (count, 1, x, y, z))
                count := count - 1
                t := y ; y := z ; z := t
                goto start
    after_1: move(x, y )
                    -- Трансляция hanoi (n-1, other, target, source):
                stack.put (create{RECORD}.make(count, 2, x, y, z))
                count := count - 1
                t := x ; x := z ; z := t
                goto start
            end
                    -- Трансляция возврата:
    after_2: if not stack.is_empty then
                top := stack.item      -- Вершина стека
                count := top.count
                x := top.source ; y := top.target ; z := top.other
```

```
            call := top.call ; stack.remove
            if call = 2 then
                  goto after_2
            else
                  goto after_1
            end
      end
            -- Отсутствует предложение else: программа завершается тогда и
            --  только тогда, когда стек пуст.
end
```

Тело процедуры *iterative_hanoi* получено из рекурсивной процедуры *hanoi* систематическим применением техники исключения рекурсии.

D1 Для каждого аргумента вводится локальная переменная. В примере используется простое соглашение о наименовании стержней: *x* для *source* и так далее.

D2 Соответствующей локальной переменной присваивается значение аргумента. Дальнейшая работа выполняется над локальной переменной. Это необходимо, поскольку процедура не может изменять значения аргументов (*n:= new_value*; – некорректно).

D3 Задать метку (здесь *start*) первому оператору исходного текста процедуры (после инициализации локальных переменных, добавленной в пункте D2).

D4 Ввести еще одну локальную переменную, здесь *call*, со значениями, идентифицирующими различные рекурсивные вызовы в теле. Здесь есть два рекурсивных вызова, так что *call* имеет два возможных значения, произвольным образом заданные как 1 и 2.

D5 Добавить метки, здесь *after*_1 и *after*_2, к операторам, непосредственно следующим за каждым рекурсивным вызовом.

D6 Заменить каждый рекурсивный вызов операторами, которые:
- вталкивают в стек запись активации, содержащую значения локальных переменных;
- локальным переменным, представляющим аргументы, присваивают значения фактических аргументов вызова; здесь рекурсивный вызов заменяет значение *n* на *n* -1 и выполняет обмен значений *other* и *target*;
- переход к началу кода.

D7 В конце процедуры добавляются операторы, которые завершают выполнение процедуры, только когда стек пуст, а в противном случае:
- восстанавливают значения всех локальных переменных из записи активации в вершине стека;
- получают из этой записи значение переменной *call*;
- удаляют запись из стека;
- переходят в нужную точку кода, зная значение *call*.

Эта общая схема применима к исключению рекурсии в любой рекурсивной программе, выполняемой как самим программистом в собственных целях, так и разработчиками трансляторов.

Мы теперь рассмотрим возможность ее упрощения, включая удаление **goto**. Нам понадобится более глубокое понимание структуры программы. В то же время убедитесь, что вы хорошо понимаете этот «грубый» вариант исключения рекурсии.

Рис. 14.30. Питер Наур и Джим Хорнинг (2006)

Почувствуй историю

Когда полагали рекурсию невозможной (рассказ Джима Хорнинга)

Летом 1961 года я пригласил прочитать лекцию в Лос-Анджелесе малоизвестного ученого из Дании. Его звали Питер Наур, и темой его лекции был новый язык программирования Алгол 60. Когда пришло время вопросов, мужчина, сидевший рядом со мной, встал и сказал: «Мне кажется, что в ваших слайдах есть ошибка».

Питер был озадачен: «Нет, я так не думаю. В каком слайде?»

«В том, на котором показано, что программа вызывает саму себя. Реализовать это невозможно».

Питер был озадачен еще больше: «Но мы же реализовали язык полностью и пропустили все примеры через наш компилятор».

Мужчина сел, но продолжал бормотать: «Невозможно! Невозможно!»

Я подозреваю, что не один он в зале думал так же.

В то время общепринятой практикой было статическое распределение памяти для кода программы, ее локальных переменных и адреса возврата. Стек вызовов, по меньшей мере, дважды независимо был придуман в Европе под различными именами, но все еще не был широко понимаем в Америке.

Говоря о независимом изобретении понятия стека вызовов, Хорнинг, видимо, имеет в виду Фридриха Бауэра из Мюнхена, который использовал термин Keller (cellar), и Эдсгера Дейкстру из Голландии, когда он реализовал свой собственный компилятор Алгола 60.

Рис. 14.31. Фридрих Бауэр (2005)

Упрощение итеративной версии

Код, данный выше, в сравнении с исходным рекурсивным вариантом выглядит громоздким. Действительно, для серьезного рекурсивного алгоритма итеративная версия никогда не будет выглядеть столь же элегантной, как рекурсивная. Но мы попытаемся приблизиться к этому эталону, пересмотрев источники компиляции.

- Мы можем заменить **goto** конструкциями структурного программирования.
- Идентифицируя обратимые преобразования аргументов, можно ограничить объем хранимой информации в записи активации, которую нужно записывать и получать из стека.
- В некоторых случаях (хвостовая рекурсия) можно вообще обойтись без стека.

Последние два случая упрощений могут быть также важны для улучшения производительности, поскольку операции над стеком требуют и времени, и памяти.

В примере с Ханойской башней первым делом устраним торчащие как заноза операторы **goto**. Чтобы абстрагироваться от лишних деталей кода, запишем тело процедуры *iterative_hanoi* в виде:

```
INIT
start: if count > 0 then
                SAVE_AND_ADAPT_1
                goto start
after_1:       MOVE
                SAVE_AND_ADAPT_2
                goto start
          end
after_2: if not stack.is_empty then
                RETRIEVE
                if call = 2 then goto after_2 else goto after_1 end
          end
```

Здесь *SAVE_AND_ADAPT_1* представляет сохранение информации в стеке и изменение значений перед первым вызовом, *SAVE_AND_ADAPT_2* – то же для второго вызова, *RETRIEVE* – получение информации из стека, включая значение *call*, *MOVE* – базисную операцию переноса, *INIT* – инициализацию локальных переменных значениями аргументов.

В параграфе 7.11 при обсуждении вопроса избавления от **goto** подобный пример уже был рассмотрен. Еще раз проанализировав его, нетрудно понять, что наша программа может быть записана с циклами, но без **goto**:

```
from INIT until over loop
    from until count <= 0 loop
        SAVE_AND_ADAPT_1
    end
    from stop := stack.is_empty until stop loop
            RETRIEVE
            stop := (stack.is_empty or (call /= 2))
    end
    over := (stack.is_empty and (call = 2))
```

```
    if not over then MOVE ; SAVE_AND_ADAPT_2 end
end
```

И этот вариант можно упростить, удалив, в частности, булевскую переменную *stop*:

```
from INIT until over loop
    from until count = 0 loop SAVE_AND_ADAPT_1 end
    from call := 2 until stack.is_empty or call = 1 loop RETRIEVE end
    over := (stack.is_empty and (call = 0))
    if not over then MOVE ; SAVE_AND_ADAPT_2 end
end
```

Упрощения являются результатом анализа возможных значений переменных.

- Так как *count* никогда не может стать отрицательной величиной из-за предусловия, требующего его положительности, и условия завершения вычислений, то вполне законно заменить тест *count* <= *0* на тест *count* = 0.
- Для избавления от stop заметим, что значение call может быть только 1 или 2, так что допустимо заменить тест *call* /= 2 на тест *call* = 1. После чего установим начальное значение *call* = 2, так что это условие будет учитываться для второй и последующих итераций, если таковые будут.

Хвостовая рекурсия

Стандартная техника, позволяющая уменьшить затраты на операции над стеком по вталкиванию и выталкиванию элементов, основана на том, что нет необходимости хранить контекстную информацию, а позже получать ее, если алгоритму эта информация больше не нужна. В частности, это происходит при рекурсивном вызове, который является последней операцией, выполняемой экземпляром рекурсивной программы.

Это упрощение применимо к примеру *hanoi*. Второй рекурсивный вызов является последним оператором, выполняемым при активации процедуры. Это означает, что нет необходимости в *SAVE_AND_ADAPT_2*, или, более точно, единственная информация, которую требуется сохранить, – это значение *call*, так как при возврате необходимо анализировать это значение.

Хороший компилятор может обнаружить хвостовую рекурсию и применить эту оптимизацию для улучшения производительности рекурсивного алгоритма.

В случае с *hanoi* устранение хвостовой рекурсии может и не проводиться, так как существует другая, более мощная оптимизация, почти полностью устраняющая необходимость в стеке, с которой мы вскоре познакомимся. Но все же следует освоить на практике оптимизацию, связанную с хвостовой рекурсией, удалив операции, в которых нет необходимости.

Преимущества обратимых функций

Использование стека для хранения значений перед вызовом и получение их после завершения вызова – это техника по умолчанию, которая работает во всех случаях; но мы уже видели ранее, что существует альтернатива данному подходу – обращение преобразования аргументов. В случае с *hanoi* обращение возможно для всех аргументов.

- Трансформация, применяемая при каждом вызове, *count*:= *count* - 1, имеет очевидное обращение: *count*:= *count* + 1.

- Для других аргументов, представляющих стержни, трансформация задается операцией взаимного обмена: $swap_{23}$ – для первого вызова и $swap_{12}$ – для второго, где $swap_{ij}$ означает операцию обмена между стержнями с номерами *i* и *j*. Понятно, что эта трансформация обратима, более того, $swap_{ij}$ является собственным обращением, поскольку двойной обмен восстанавливает исходное состояние.

Так что, фактически, нет необходимости хранить в стеке ни *count*, ни *x, y, z*. Достаточно при достижении *RETRIEVE* выполнять соответствующее обращение:

```
"Получение значения call"
count := count + 1
if call = 1 then swap23 else swap13 end
```

Стек остается необходимым, но только для записи и получения *call*. Упрощение становится еще существеннее, если вспомнить, что *call* имеет только два значения: 1 и 2. Но ничто не мешает нам изменить соглашение и рассматривать их как булевские значения 1 и 0. Тогда можно применить стек, содержащий булевское значение. Более того, если допустимо ограничить высоту стека, то вместо стека можно использовать единственную *целочисленную переменную*, скажем, *s* (в современных компьютерах целочисленные переменные могут иметь длину в 64 бита). Тогда операции над стеком моделируются операциями над целым *s*, рассматриваемым как строка битов:

```
s = 1            – Пуст ли стек?
s := 1           – Инициализация пустого стека
s := 2*s         – Втолкнуть 0 (сдвиг влево на один разряд строки битов)
s := 2*s + 1     – Втолкнуть 1 (сдвиг влево на один разряд строки битов с
                 – приписыванием 1)
b := s \\ 2      – Получить (в b) значение с вершины стека
                 – (\\ остаток от деления нацело)
s:= s // 2       – Удалить значение с вершины стека
                 – (// - деление нацело - сдвиг вправо)
```

Вот результат выполнения некоторой последовательности таких операций.

Оператор	Цель	Результат	Бинарное представление s (часть нулей слева опущена)
s:= 1	— Начать с пустого стека	*s* = 1	1
s:= 2**s*	— Втолкнуть 0	*s* = 2	10
s:= 2**s* + 1	— Втолкнуть 1	*s* = 5	101
s:= 2**s* + 1	— Втолкнуть 1	*s* = 11	1011
s:= 2**s*	— Втолкнуть 0	*s* = 22	10110
s:= *s* // 2	— Вытолкнуть	*s* = 11	1011
b:= *s* \\ 2	— Прочитать элемент вершины	*b* = 1	

В последнем столбце показано бинарное представление целого. Если нумеровать разряды в этом представлении справа налево, начиная с 0, то единица в разряде *k* имеет значение 2^k. Значение 0 в самом правом разряде означает, что число четное, 1 – нечетное. Когда такое

представление задает стек булевских значений, вершиной стека является самый правый разряд. Пустой стек задается значением *s*, равным 1.

Техника использования единственного целого числа для задания стека булевских значений может безопасно использоваться, когда гарантируется, что размер стека не превосходит длины целого в битах. В примере с *Hanoi* проблемы не возникает, поскольку 2^{63} или даже 2^{31} – число, задающее количество ходов, столь велико, что компьютеру не справиться с вычислениями за разумное время.

В результате обсуждений приходим к более простой и эффективной форме алгоритма *iterative_hanoi* с аргументами *n, source, target, other*:

```
from
    count := n ; x := source ; y := target ; z := other ; s := 1
until over loop
    from until count = 0 loop
        swap23 ; s := 2*s + 1 ; count := count - 1
    end
    from call := 0 until s = 1 or call = 1 loop
        call := s \\ 2 ; s := s // 2 ; count := count + 1
        if call = 1 then swap23 else swap13 end
    end
    over := ((s = 1) and (call = 0))
    if not over then
        move (x, y)
        swap13 ; s := 2*s ; count := count - 1
    end
end
```

Хотя полученная программа является результатом систематических трансформаций, а не программой, которую вы бы написали с самого начала (рекурсия яснее и проще), интересно проследить за ее выполнением, сравнивая с оригинальной рекурсивной версией, а особенно – с бинарным деревом выполнения, представленным в начале этой главы и показывающим выполнение как инфиксный обход дерева:

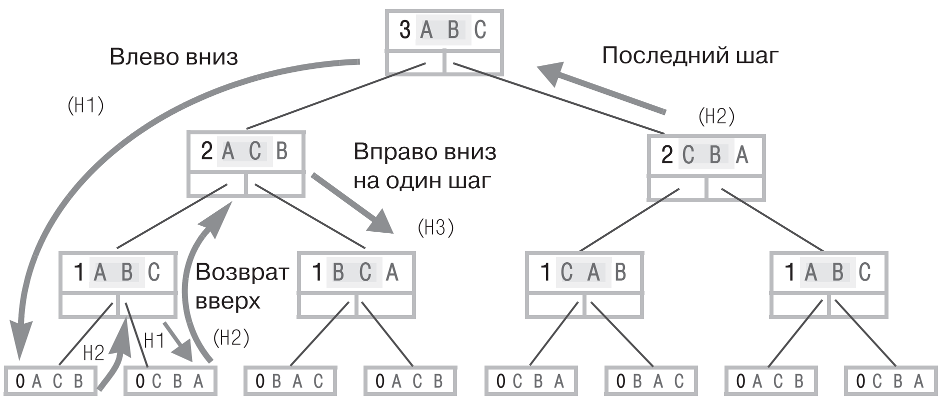


Рис. 14.32. Обход бинарного дерева задачи Hanoi

В алгоритме можно выделить три компонента.

H1 Самый левый в глубину – идти насколько возможно вниз, влево, пока не достигнешь листа. У листьев значение $n = 0$ (*count* в этой версии), хотя на предыдущих рисунках дерево заканчивалось на 1, поскольку на нулевом уровне ничего не происходит.

H2 Возврат вверх. Если вы возвращаетесь из правого поддерева, то продолжаете идти вверх, поскольку это означает завершение второго рекурсивного вызова, а следовательно – и завершение работы текущего экземпляра процедуры.

H3 Поднявшись вверх по левой ветви, выполняем посещение корня – перенос диска из x на y, а потом идем вниз по правой ветви.

Все это повторяется, пока, придя справа (H2), не обнаружим, что стек пуст.

При спуске вниз (H1, H3) уменьшается *count* и выполняется обмен y и z, если идем слева (H1), и обмен x и z, если идем справа (H3). При возврате назад (H2) восстанавливаются исходные значения, увеличивая *count* и выполняя подходящий обмен в зависимости от того, справа или слева вы пришли. Анализ вершины стека, хранящей значение *call*, позволяет понять, откуда мы пришли в узел – слева (первый вызов) или справа (второй вызов).

14.10. Ключевые концепции, рассмотренные в этой главе

- Часто удобно определять понятие рекурсивно. Это означает, что определение понятия использует один или несколько экземпляров самого понятия.
- Чтобы такое определение было полезным, любое вхождение понятия должно применяться к меньшей цели в сравнении с исходной. Необходимо также существование нерекурсивной ветви, что позволяет, в конечном счете, любое применение определения свести к конечной комбинации элементарных вариантов.
- Рекурсивные определения, в частности, могут быть полезными при определении программ, структур данных и грамматик.
- Любой цикл может быть записан в эквивалентной рекурсивной форме, используя простую трансформацию.
- Справедливо и обратное. Любой рекурсивный алгоритм имеет свободный от рекурсии эквивалент, но трансформация нужна более изощренная. Она требует изменения потока управления, сохранения локальной информации о каждом рекурсивном вызове, так же как и получение ее в процессе дальнейшей работы. Эта трансформация предполагает работу со стеком или использование обратимых трансформаций данных.

Новый словарь

Activation	Активация	**Activation record**	Запись активации
Alpha-beta	Альфа-бета	**Backtracking**	Перебор с возвратами
Binary tree	Бинарное дерево	**Call chain**	Цепочка вызовов
Depth-first	Первый в глубину	**Direct recursion**	Прямая рекурсия
Indirect recursion	Непрямая (косвенная) рекурсия	**Inorder**	Инфиксный
		Iterative	Итеративный
Instance (of a routine)	Экземпляр (программы)	**Minimax**	Минимакс
		Non-creative	Не творческий
Postorder	Постфиксный	**Preorder**	Префиксный
Recursion	Рекурсия	**Recursive**	Рекурсивный
Recursive definition	Рекурсивное определение	**Traversal**	Обход

14-У. Упражнения

14-У.1. Словарь

Дайте точные определения терминам словаря.

14-У.2. Не слишком ли много рекурсии?

Является ли определение «рекурсивного определения» рекурсивным?

14-У.3 Бинарные деревья поиска с повторениями

Для каждого приведенного в этой главе метода поиска в бинарных деревьях перепишите объявление (если требуется), допуская возможность множественного вхождения элемента *item* в дерево.

14-У.4. Язык программирования без программных текстов

Напишите компилятор и интерпретатор элементарного языка программирования. Используйте приемы, обсуждаемые в предыдущих главах. Решение должно использовать рекурсию. Чтобы избежать проблем с конкретным синтаксисом, ваш инструментарий должен иметь дело непосредственно со структурами данных, а не с текстом программы.

Наш маленький язык, назовем его АСТ («Абстрактный синтаксис только»), имеет следующие свойства.

- Единственный тип данных – integer.
- Все переменные принадлежат типу integer. Они не объявляются. Имя переменой – любая строка символов.
- Разрешается использовать целочисленные константы, например, 123.
- Выражения формируются из констант, переменных скобок и четырех операций – сложение, вычитание, умножение и деление нацело.
- В языке два вида операторов – присваивание и печать.
- Программа на АСТ состоит из последовательности присваиваний и последовательности операторов печати, каждая из которых может отсутствовать.
- Выполнение программы состоит из инициализации нулями переменных программы, выполнении последовательности присваиваний и последующей печати значений переменных.

Типичная программа на АСТ приведена здесь с учетом конкретного синтаксиса, хотя он и не является частью определения языка:

```
assign
    x := 3
    y := 5
    x := 2*(x + (y // 3))
then
    print x
    print z
end
```

В результате выполнения этой программы должно быть напечатано одно значение – 8.

Синтаксис конкретной программы является одним из возможных выборов. Вполне возможно, например, вместо слова **then** использовать ключевое слово **print**. Печать переменных можно задавать списком без повторения **print**.

Напишите множество классов, включающее *PROGRAM, ASSIGNMENT, PRINT, EXPRESSION*. Методы этих классов, включающие процедуры создания, должны позволять построить абстрактное синтаксическое дерево, задающее АСТ-программу.

1. Добавьте класс с процедурой, которая использует эти классы и их методы для создания абстрактного синтаксического дерева, представляющего программу нашего примера.
2. Добавьте в класс *PROGRAM* процедуру *write_out*, которая выполняет текстуальное представление АСТ-программ в том виде, как оно дано в примере. Выполните программу из шага 2 и убедитесь в корректности полученного результата. *Подсказка*: вам необходима рекурсивная процедура обхода, подобная той, которая рассматривалась в данной главе.
3. Напишите АСТ-интерпретатор в форме процедуры *interpret* в классе *PROGRAM*, который выполняет программу и вырабатывает ожидаемый результат. Запустите ее на данном примере и проверьте результат выполнения.
4. Напишите АСТ-Eiffel-компилятор в форме процедуры *compile* в классе *PROGRAM,* которая АСТ-программу преобразует в программу на Eiffel, сохраняя семантику АСТ-программ. Корневой класс с подходящей процедурой создания и другие классы необходимы для решения этой задачи. Используя Eiffel-студию, выполните наш пример и проверьте результат.

Терминологическое замечание. Результатом шага 5 является *реанализатор – unparser*, который создает текст программы по внутреннему представлению, такому как абстрактное синтаксическое дерево, выполняя операцию, обратную тому, что делает классический *анализатор – parser*.

14-У.5. Вставка без рекурсии

Напишите версию *put* для бинарного дерева поиска, используя цикл, а не рекурсию.

Подсказка: источником вдохновения может служить реализация метода *has*.

14-У.6. Рекурсивный реверс

Сохраняя предположения (список остановок известен своей первой ячейкой типа *STOP*, остальные остановки доступны через повторное применение *next*), перепишите функцию *reversed*, используя рекурсию вместо цикла (смотри также следующее упражнение).

14-У.7. Реверс списка. Функциональный стиль

Напишите рекурсивную функцию для обращения связного списка (аргумент и результат должны быть типа *LINKED_LIST*[*G*]). Сведите к минимуму манипуляции с указателями и приблизьтесь, насколько возможно, к стилю функции *reversed*, приведенному как пример программирования на Haskell. Проанализируйте временную и емкостную сложность вашего решения.

14-У.8. Сокращение перебора с возвратами

Адаптируйте общий алгоритм перебора с возвратами так, чтобы он сохранял историю ранее исследованных позиций и удалял любой путь, ведущий к такой позиции. Можете предположить, что *PATH* имеет запрос *position*, определяющий терминальную позицию пути.

14-У.9. Игнорирование циклов

Адаптируйте общий алгоритм перебора с возвратами так, чтобы он не исследовал пути длиннее, чем *path_cutoff* – заданное целое число.

14-У.10. Свойства графа функции

(Это упражнение не требует программирования, но предполагает проведение математического анализа)

Для последовательных аппроксимаций H_i графа функции, связанной с Ханойской башней (параграф 14.7: «Башни снизу вверх»), определите:

1. Каково число пар в H_i?
2. Задайте математическую формулу для H_i.

14-У.11. Программирование графа функции снизу вверх

1. Спроектируйте класс, каждый экземпляр которого задает пару «аргумент-результат» в форме [(*n, s, t, o*),<...>] для графа функции, связанной с Ханойской башней.
2. Основываясь на классе из пункта 1, спроектируйте класс, представляющий граф функции в целом.
3. Из этих классов и правил [5] и [6] (параграф 14.7: «Башни снизу вверх»), определяющих граф функции в интерпретации рекурсии «снизу вверх», напишите программу, которая для любого i вычисляет i-ю аппроксимацию графа H_i. Алгоритм может использовать циклы, но не может использовать рекурсию.
4. Используйте эту программу для печати последовательности ходов (с источником ‘A’ и целью ‘B’) для нескольких значений i. Убедитесь, что результаты соответствуют работе рекурсивной процедуры.

14-У.12. Алгоритмы бинарного дерева с точки зрения «снизу вверх»

Рассмотрим рекурсивный алгоритм обхода бинарного дерева: вы можете выбрать префиксный, инфиксный или постфиксный порядок обхода.

1. Спроектируйте модель, которая интерпретирует обход как функцию, возвращающую последовательность узлов. Источником вдохновения может служить анализ «снизу вверх» для Ханойской башни.
2. Напишите рекурсивное «определение» этой функции.
3. Выразите это «определение» в виде уравнения неподвижной точки на графе функции, используя T_i, как имя графа для бинарного дерева высоты i.
4. Используйте это определение для создания (либо вручную, либо написав небольшую программу) T_5 для примера бинарного дерева и результирующего порядка обхода.

14-У.13. Рекурсия без оптимизации

(Это упражнение требует доступа к компилятору, например, C или C++, с поддержкой оператора **goto**)

Реализуйте и протестируйте прямую итеративную трансляцию процедуры *hanoi* в ее начальном варианте, используя **goto** и стек без оптимизации.

14-У.14. Сохранение стека сохранения

1. Реализуйте и протестируйте итеративную, без **goto**, основанную на стеке версию Ханойской башни.
2. Улучшьте решение, используя оптимизацию, основанную на хвостовой рекурсии, избегая во втором вызове ненужного сохранения данных.
3. При условии, что выполнено предыдущее упражнение, примените ту же оптимизацию к версии с **goto**.

14-У.15. Обход без стека

Мы видели, что реализация рекурсии требует обращения преобразования аргументов рекурсивного вызова. Стек является одним из возможных путей решения этого требования. Используя подходящие приемы обращения, реализуйте обход бинарного дерева, например, в инфиксном порядке, без рекурсии и без стека, за исключением, возможно, стека булевских значений (или, эквивалентно, бита в каждом узле).

Подсказка: временно переопределите связи дерева, сохраняя информацию о том, откуда пришли в узел.

Контр-подсказка: решение можно найти, набрав при поиске в Интернете слова *Deutsch*, *Shorr* или *Waite* (имена авторов известного алгоритма, основанного на этой идее). Не делайте этого! Спроектируйте алгоритм самостоятельно, затем посмотрите ссылки, если пожелаете.

14-У.16. Транзитивное замыкание

(Это упражнение ссылается на последнюю главу)

Сформулируйте определение транзитивного замыкания как рекурсивное определение.

14-У.17. Матричная алгебра для продукций БНФ

(Это упражнение требует знания основ линейной алгебры)

Рассмотрим БНФ-продукции – небольшой пример из этой главы или более расширенный из предыдущих глав, включающий только продукции для конкатенации и выбора (без повторения, поскольку оно может быть заменено комбинацией двух других).

1. Рассматривайте конкатенацию лексем как «умножение», а альтернативный выбор – как «сложение». Покажите, что в этом случае возможно выразить грамматику как матричное уравнение $X = A*X + B$, где X – это вектор нетерминалов, A – матрица из терминалов и нетерминалов, и B является вектором.
2. Обсудите пути решения такого уравнения, следуя модели, предложенной для уравнения неподвижной точки.

15

Проектирование и инженерия алгоритма: топологическая сортировка

Одна из тех радостей, что доставляет нам изучение информатики, – это знакомство с замечательными алгоритмами. В этой главе мы займемся исследованием алгоритма, который обладает рядом достоинств, заслуживающих нашего внимания. Он полезен на практике, он, обладая хорошим математическим базисом, элегантен, он иллюстрирует технику решения задач, применимую в различных контекстах.

Познакомить вас с уже готовым решением неинтересно. Вместо этого мы начнем разрабатывать это решение шаг за шагом, как это делается для всякой научной проблемы. Мы начнем с постановки задачи, перейдем к ее математическому анализу, займемся поиском структуры данных, обеспечивающих корректность и эффективность решения. Мы не только спроектируем алгоритм, но, не останавливаясь на этом, получим полное *инженерное* решение, удовлетворяющее всем практическим требованиям. В конце главы уроки этого примера позволят сформулировать некоторые общие принципы разработки алгоритмов и инженерии программ.

15.1. Постановка задачи

Вы решили сегодняшний день посвятить культуре: что может быть лучше, чем посещение Лувра и музея Орсе в Париже? Но для этого вам понадобится карта, и нужно приобрести проездной на метро, поскольку срок старого уже истек, но для проездного нужны деньги, так что придется зайти в банк или воспользоваться банкоматом. Можно выразить эти ограничения следующим образом:

```
[Map, Louvre], [Map, Orsay], [Pass, Louvre], [Pass, Orsay], [Money, Pass]
```

Здесь каждое ограничение задается парой [*x*, *y*], означающей, что «*x* должно случиться прежде чем *y*». Это можно отобразить графически:

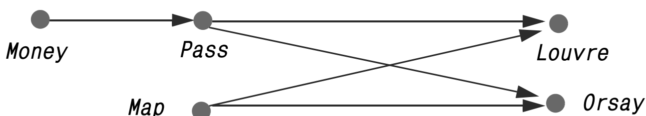


Рис. 15.1. Упорядочение ограничений

Топологической сортировкой множества элементов, подчиняющихся ограничениям порядка, называется перечисление элементов в порядке, удовлетворяющем ограничениям. В данном примере топологическими сортировками являются последовательности:

```
Money, Pass, Map, Louvre, Orsay
Map, Money, Pass, Orsay, Louvre
Money, Map, Pass, Louvre, Orsay
```

Заметьте: последовательность *Pass, Money, Map, Louvre, Orsay* некорректна, поскольку нарушается ограничение [*Money, Pass*].

Задача топологической сортировки может иметь:

- несколько решений, как в данном примере;
- в точности одно решение;
- ни одного решения, как было бы, в случае если и только если ограничения образовывали бы цикл – множество ограничений: [e_1,e_2], [e_2, e_3], ...[e_n, e_1] для некоторого $n >= 1$. Если мы добавим в наш пример ограничение [*Orsay*, *Money*], создавая тем самым цикл, то никакого решения больше существовать не будет, так как мы, с одной стороны, требуем, чтобы «утром – стулья, вечером – деньги», а с другой стороны, требуем «деньги вперед».

Если существует более одного решения, то возникает проблема выбора одного из них. Обычно с любым решением связывается некоторая *стоимость*, тогда целью может быть поиск решения с минимальной стоимостью. Вскоре при анализе алгоритма мы увидим, где может применяться этот критерий при поиске нужного решения. Другой подход может состоять в том, чтобы найти *все* решения.

Примеры применения

Задача топологической сортировки возникает всякий раз, когда требуется упорядочить элементы при некоторых ограничениях на их порядок следования. Вот несколько примеров.

- Для графического дисплея рассмотрим окна – графические прямоугольники, которые могут пересекаться. Нам необходим алгоритм, рисующий эти прямоугольники с учетом их наложения друг на друга. Это задача топологической сортировки. Зададим следующие ограничения: [B, A], [D, A], [D, C], [B, D], [E, C]. Здесь ограничение [*x, y*] содержательно означает «*x* не должно скрывать никакую часть *y*». Возможным решением является последовательность: B, D, E, A, C.

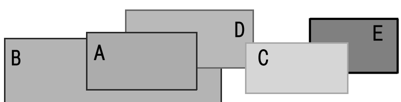


Рис. 15.2. Прямоугольники с ограничением предшествования

- При промышленной сборке сложных изделий – турбины или самолета – множество задач по управлению сборкой сопровождается ограничениями, например, структурная работа над элементом должна предшествовать его окраске. Топологическая сортировка дает расписание, совместимое с этими ограничениями.
- Подобная задача возникает при управлении проектами, особенно при управлении программными проектами. Если проект связан с некоторой проблемной областью, то он должен сопровождаться глоссарием (словарем) технических терминов (непонима-

ние между экспертами проблемной области и разработчиками ПО является главным источником ошибок и провалов). Определение любого термина может ссылаться на другие термины. Термины в словаре могут идти в алфавитном порядке, как принято в словарях, но также полезно иметь версию словаря, которая может читаться в последовательности, где каждый термин объясняется через уже определенные термины. Создание такого словаря предполагает топологическую сортировку.

- При описании класса вы, возможно, хотели бы, чтобы его компоненты шли в порядке, гарантирующем, что никакой вызов компонента не встречается до его описания[1].
- Топологическая сортировка позволяет улучшить эффективность компиляции ОО-программ при реализации наследования и в особенности – динамического связывания, обсуждаемого в последующих главах. Проблема актуальна для больших программ со многими классами и состоит в нумерации классов таким образом, чтобы номер, приписанный классу, был близок к номерам классов, являющихся его потомками – классами, которые наследуют от него прямо или косвенно. Используя «наследует от» как отношение порядка, компилятор EiffelStudio, основываясь на топологической сортировке, достигает существенной оптимизации требуемой памяти, что является основой жизнеспособности ОО-подхода.

Точки на плоскости

Вот пример, дающий удобную визуализацию задачи. Рассмотрим точки на плоскости:

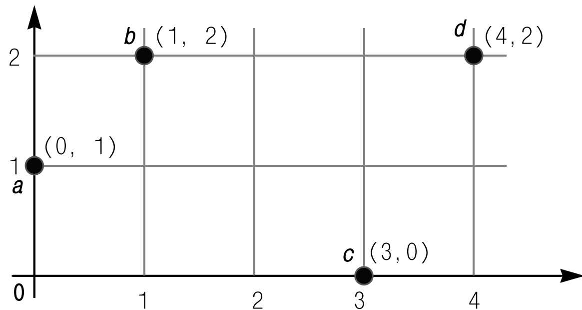


Рис. 15.3. Конечное множество точек

Введем для точек p_1 с координатами (x_1, y_1) и p_2 с координатами (x_2, y_2) отношение << («строго меньше»), устанавливающее, что $p_1 << p_2$, если:

- $x_1 \leqslant x_2$;
- $y_1 \leqslant y_2$;
- $p_1 \neq p_2$ (точки не совпадают).

Для четырех точек, показанных на рисунке, справедливо следующее:

a<<b	a<<d	b<<d	c<<d

1 В некоторых языках программирования таково требование синтаксиса. Но взаимная рекурсия методов, когда A вызывает B, а B вызывает A, приводит к циклу, так что в язык приходится вводить специальные конструкции предварительного объявления (предварительно объявляется B, затем объявляется A, затем дается полное объявление B).

Топологическая сортировка для этого отношения дает любое перечисление точек, для которого если $p << q$, то в списке перечисления p предшествует q. Для наших четырех точек существуют три таких перечисления:

```
a, b, c, d
a, c, b, d
c, a, b, d
```

Три возможных обхода точек можно визуализировать:

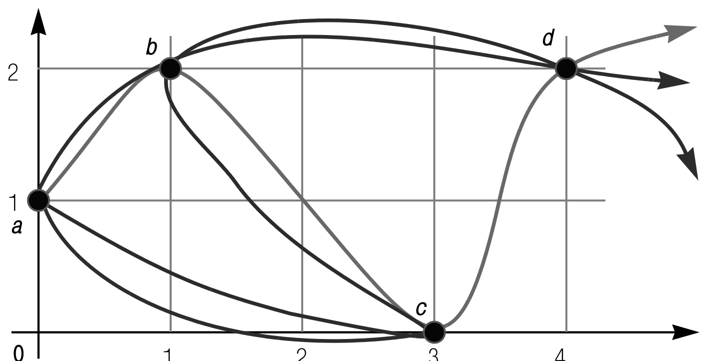


Рис. 15.4. Три типологических сортировки множества точек

С другой стороны, перечисление *a, d, b*, c не совместимо с отношением, поскольку свойство $c << d$ требует, чтобы c предшествовало d.

15.2. Основы топологической сортировки

Задача, обсуждаемая в этой главе, имеет точную математическую формулировку.

Определение: задача топологической сортировки

> Для заданного на конечном множестве ациклического отношения *r* требуется найти отношение полного порядка, для которого *r* является подмножеством.

Это определение использует математические понятия – отношения как множества, ациклические отношения, отношения порядка (полного или частичного), которые теперь предстоит рассмотреть.

Бинарные отношения

Определение: отношение

> **Отношение** на множестве *A* (для простоты рассматривается только бинарное отношение) задается множеством пар в форме [*x, y*], где оба элемента пары являются элементами *A*.

Примером отношения на множестве {1, 2, 3}является:

```
{[1, 2], [1, 3], [2, 3]}
```

Мы можем назвать это отношением < («меньше», так как *x* и *y* принадлежат множеству {1, 2, 3} и *x* меньше *y*).

Мы можем использовать отношения для описания предыдущих примеров.

- Отношение *below* (ниже) на множестве прямоугольников, содержащее все пары прямоугольников [*x, y*], таких, что в области наложения прямоугольников должны показываться точки *y*, а не *x*.
- Отношение *before* (прежде) – множество пар {[*Map, Louvre*], [*Map, Orsey*], ...} – является отношением на множестве {*Pass, Money, Map, Louvre, Orsay*}, содержащим пары [*x, y*], такие, что событие *x* должно произойти раньше *y*.
- Отношение *used_in* («используется в»), заданное на множестве терминов словаря, содержит все пары [*x, y*], такие, что определение термина *y* содержит термин *x*.
- Отношение *called_by* («вызывается в»), заданное на множестве методов класса, содержит все пары [*x, y*], такие, что тело метода *y* содержит вызов метода *x*.
- Отношение << над точками – множество пар {[*a, b*], [*a, d*], [*b, d*], [*c, d*]}.

Ациклические отношения

Все наши примеры были до сих пор ациклическими отношениями. Это понятие определяется следующим образом:

Определение: ациклическое отношение

Отношение является **ациклическим**, если у него нет циклов.

Это определение следует дополнить.

Определение: цикл в отношении

Циклом для отношения *r* над множеством *A* является последовательность x_1, ... x_m ($m \geqslant 2$) из элементов *A*, такая, что все последовательные пары [x_i, x_{i+1}] для $1 \leqslant i < m$ принадлежат *r*, и $x_m = x_1$.

Отношение *before*, введенное ранее, циклов не имеет:

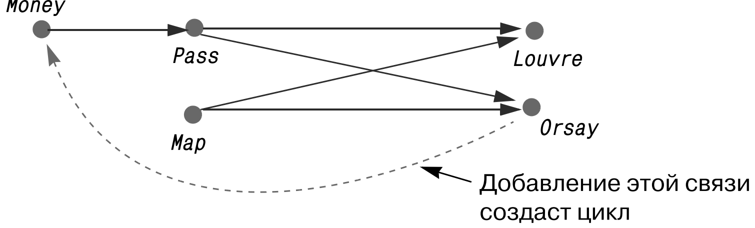


Рис. 15.5. Отношение, описывающее ограничение порядка

Простейший случай цикла для отношения *r* встречается (при *m* = 2) для элемента *x*, такого, что [*x, x*] принадлежит *r*.

Для успеха топологической сортировки требуется ациклическое отношение, хотя мы будем рассматривать алгоритм, который может частично обрабатывать отношение, включающее цикл. Если множество, на котором определено отношение, конечно, то ациклическое отношение имеет важное свойство, критичное для алгоритма топологической сортировки.

Теорема об отсутствии предшественника

Для любого ациклического отношения *r* над непустым, конечным множеством *A* существует элемент *x* из *A*, не имеющий предшественника в *r*.

Определение: предшественник

Предшественником элемента *y* для отношения *r* является элемент *x*, такой, что пара [*x, y*] принадлежит *r*.

Теорема доказывается от противного. Предположим противное, что каждый элемент в *A* имеет по меньшей мере одного предшественника. Пусть x_1 – элемент из *A* (он существует, так как множество не пусто). По предположению, у него есть хоть один предшественник, выберем любой и назовем его x_2. По тем же причинам есть предшественник x_3 и у x_2, так что в *r* существует пара [x_3, x_2]. Продолжая вывод, следует признать существование бесконечной последовательности [x_{i+1}, x_i], принадлежащей r для каждого $i >= 1$. Поскольку *A* – конечное множество, последовательность должна иметь повторяющиеся элементы. Более точно: у последовательности x_1, x_2, ... x_{n+1}, где *n* – число элементов *A*, все элементы не могут быть различными; поэтому должны быть целые *i* и *j*, для $1 \leqslant i < j \leqslant n + 1$, такие, что $x_i = x_j$, но тогда последовательность x_j, x_{j-1}, ... x_i является циклом. Пришли к противоречию.

Доказательство конструктивно, и мы будем его использовать при проектировании алгоритма топологической сортировки. Для выработки перечисления алгоритм будет выискивать на каждой итерации элемент, не имеющий предшественника в оставшемся отношении порядка.

Предположение о конечности множества *A* существенно. Теорема неприменима к бесконечным множествам, например, отношение «меньше» для целых чисел в математике ациклическое, но каждый его элемент имеет предшественника.

Отношения порядка

Идея топологической сортировки встраивает данное ациклическое отношение в отношение полного (тотального) порядка. Для определения этого понятия предварительно определим простое отношение порядка.

Определение: отношение порядка (строгое, возможно частичное)

Отношение является отношением **порядка**, если оно удовлетворяет следующим свойствам для любых элементов *x, y, z* из множества *X*, на котором оно определено.

O1 **Антирефлексивность**: отношение не имеет пар вида [*x,x*].

O2 **Транзитивность**: из того, что отношению принадлежат пары [*x, y*] и [*y, z*], следует, что отношению принадлежит и пара [*x, z*].

Такое отношение порядка является также:

О3 **асимметричным**: из того, что отношению принадлежит пара [*x, y*] следует, что отношению не принадлежит пара [*y, x*].

(Доказательство. Если бы обе пары принадлежали отношению, то из транзитивности следовала бы рефлексивность – существование пары [*x, x*], что противоречит свойству антирефлексивности)

> Полное имя для отношения порядка – строгое, возможно частичное отношение порядка. Наши отношения порядка (вскоре мы познакомимся с полным или тотальным отношением) являются строгими, в том же смысле, как отношение < – «строго меньше, чем». Также возможно работать с нестрогими версиями отношения, такими как отношение <= («меньше или равно»).

Отношение « < » на {1, 2, 3} (или на любом другом множестве целых) является отношением порядка. Таким же является отношение << на точках. Наши другие ациклические отношения – *before* для задач, *used_in* для терминов, *called_by* для методов – антирефлексивны и асимметричны, но они не обязательно транзитивны, так что они не являются отношениями порядка. Вскоре мы увидим, как можно получить транзитивную версию.

Отношения порядка в сравнении с ациклическими отношениями

Отношения порядка тесно связаны с ациклическими отношениями. В одном направлении эта связь – прямая.

Теорема: «Ацикличность и отношение порядка» (1)

Любое отношение порядка (более того, любое его подмножество) ациклично.

Доказательство от противного. Предположим, существует цикл x_1, x_2, ... x_n, где x_n то же, что и x_1. По транзитивности это влечет рефлексивность, что противоречит антирефлексивности.

> Это обобщает доказательство асимметрии: невозможность иметь обе пары [*x,y*] и [*y, x*]. Такой случай является частным случаем цикла с двумя элементами. Подобно, пара [*x, x*], нарушающая антирефлексивность, является циклом с одним элементом.

Здесь еще одно интересное свойство: транзитивное замыкание ациклического отношения является отношением порядка. Неформально транзитивное замыкание отношения представляет собой отношение порядка, полученное применением свойства транзитивности с помощью расширения исходного отношения настолько, насколько это возможно.

Это может быть проиллюстрировано на отношении *before*, выражающего ограничения порядка между задачами. Отношение антирефлексивно, асимметрично и ациклично, но оно не транзитивно, поскольку содержит пары [*Money, Pass*] и [*Pass, Louvre*], но не содержит пары [*Money, Louvre*]. Мы можем расширить отношение до транзитивного, добавив все пары вида [*x, z*], для которых в исходном отношении есть пары [*x, y*] и [*y, z*], выполняя эту операцию, пока добавление не перестанет быть возможным. Результатом этого процесса является транзитивное замыкание исходного отношения. Для нашего примера добавятся ровно две связи:

Money
Pass
Louvre
Map
Orsay
Связи, добавленные для транзитивности
Исходные связи

Рис. 15.6. Транзитивное замыкание ограничений порядка

Для отношения *called_by* между процедурами транзитивное замыкание является отношением, которое выполняется между x и y, если y вызывает x прямо или косвенно. Для отношения *child* между людьми, обозначающего множество пар $[x, y]$, таких, что x является потомком y, транзитивное замыкание служит отношением, связывающим двух людей, таких, что один из них является потомком, непосредственным или косвенным – праправнуком. Транзитивное замыкание отношения r обозначается как r^+, так что мы можем полагать, что $child^+$ = d*escendant* (потомок представляет транзитивное замыкание отношения «потомок»).

Определение: транзитивное замыкание отношения

> Транзитивным замыканием отношения r на множестве A является отношение, содержащее все пары вида $[x_1, x_m]$ для некоторой последовательности элементов $x_1, x_2, \ldots x_m$ ($m >= 2$), такой, что пары $[x_i, x_{i+1}]$ для $1 <= i <=$ m принадлежат r.

Транзитивное замыкание высвечивает другую сторону отношения между ацикличностью и порядком.

Теорема: «Ацикличность и отношение порядка» (2)

> Транзитивное замыкание любого ациклического отношения является отношением порядка.

Доказательство. Транзитивное замыкание любого отношения r очевидно транзитивно. Так что нам остается доказать антирефлексивность. Предположим, что это не так и существует элемент x, такой, что пара $[x, x]$ принадлежит r^+. По определению транзитивного замыкания существует последовательность $x_1, x_2, \ldots x_m$ ($m >= 2$), такая, что все пары $[x_i, x_{i+1}]$ для $1 <= i <= m$ принадлежат r и что x_1 и x_m равны x. Но это означает существование цикла для r. Пришли к противоречию.

Этот результат позволяет нам рассматривать ациклическое отношение как «ядро» отношения порядка. Взятие транзитивного замыкания дает нам настоящее отношение порядка. Это соответствует интуитивному пониманию отношения, например, такого как *before* между стоящими перед нами задачами, – если задача *Money* должна предшествовать задаче *Pass*, а задача *Pass* должна предшествовать задаче *Louvre*, то мы понимаем, что задача *Money* должна предшествовать задаче *Louvre*. Другими словами, мы инстинктивно принимаем транзитивное замыкание. Но когда готовятся входные данные для задачи расписания или для другой задачи, в которой предполагается топологическая сортировка, то, естественно, хотелось

бы перечислить только базисные ограничения, а не полное транзитивное замыкание. Вот почему топологическая сортировка может использовать ациклическое отношение на входе (многие представления топологической сортировки начинают с отношения порядка, но это избыточное требование, более строгое, чем требуется).

Вычисление транзитивного замыкания является «дорогой» вычислительной операцией, но нам не потребуется выполнять ее явно – алгоритм топологической сортировки будет работать с ациклическим отношением.

Тотальный порядок

Для описания выхода топологической сортировки нам понадобится уточнение понятия отношения порядка, приводящее к полному или тотальному отношению порядка. Для конечного множества полный порядок можно было бы рассматривать просто как нумерацию всех элементов множества, в котором каждый элемент появляется один раз. Но концепция полного порядка более широко применима.

Определение: отношение тотального порядка (строгое)

Тотальным порядком является отношение порядка, обладающее дополнительным свойством.

O4 Для любых *a* и *b* имеет место одно из следующих свойств:

- [*a*, *b*] принадлежит *r*;
- [*b*, *a*] принадлежит *r*;
- *a* = *b*.

Для понимания O4 заметьте: из асимметрии (O3) следует, что только одна из первых двух возможностей может существовать. Из антирефлексивности (O1) следует, что последняя возможность исключает две первые. Так что только одна из трех возможностей *может* существовать. Новое условие добавляет, что одна из трех возможностей *должна* существовать.

Отношение «<» на целых является тотальным отношением. Но не каждое отношение порядка является таковым. Отношение «<<» на точках плоскости таковым не является, так как это означало бы, что для любых двух различных точек плоскости *p*1 и *p*2 должно выполняться *p*1 << *p*2 или p2 << *p*1.

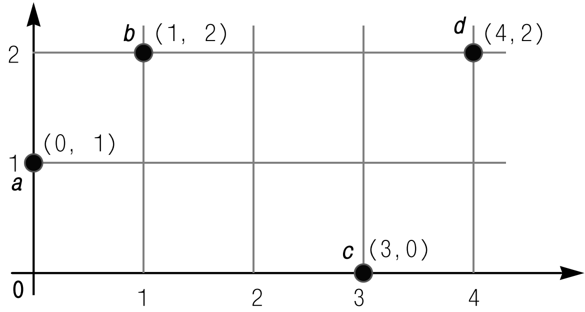


Рис. 15.7. Точки на плоскости и отношение полного порядка

Заметьте, пара точек [*a, c*] не принадлежит отношению, так как не верно, что *a* << *c*; точно так же неверно, что и *c* << *a*. Другим контрпримером является пара точек [*b, c*].

На множестве четырех точек можно установить различные отношения порядка, являющиеся полными. В частности, любое перечисление, в котором каждая точка появляется ровно один раз, можно рассматривать как задание полного порядка *t*. Вот как можно определить этот порядок: пара точек [*p, q*] принадлежит *t*, если и только если *p* предшествует *q* в перечислении. Например, перечисление [*a, b, c, d*] определяет полный порядок:

```
{[a, b], [a, c], [a, d],                                  [1]
    [b, c], [b, d],
                      [c, d]}
```

Верно и обратное: каждый полный порядок определяет единственную нумерацию.

Такой тотальный порядок является топологической сортировкой исходного отношения «<<», если и только если тотальный порядок *совместим* с исходным. Совместимость означает, что когда *p* << *q* в исходном порядке, то *p* предшествует *q* в тотальном порядке.

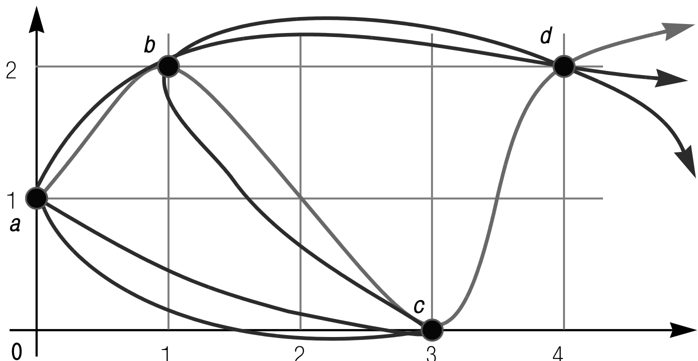


Рис. 15.8. Точки на плоскости и отношение <<

Мы уже видели, что три тотальных порядка удовлетворяют этому требованию для нашего примера: соответствующими перечислениями являются *a, b, c, d* (в [1] представлено в виде множества пар); *a, c, b, d* и *c, a, b, d*.

Что означает «совместимость», если это понятие рассматривать более точно? Его нетрудно специфицировать, благодаря определению отношения как множества пар. Сказать, что тотальный порядок, заданный перечислением *a, b, c, d*, совместим с заданным ациклическим отношением, – это все равно, что сказать, что множество пар этого отношения является подмножеством множества пар тотального отношения. Каждая пара в отношении порядка является парой в тотальном отношении порядка. В нашем примере отношение << является множеством пар:

```
{[a, b], [a, d], [b, d], [c, d]}                          [2]
```

Это множество действительно является подмножеством множества пар [1], задающего тотальный порядок. Это свойство выражает тот факт, что когда ограничение задает некоторый порядок между двумя элементами, то на выходе алгоритм должен перечислять эти элементы в заданном порядке.

Так вырабатывается определение топологической сортировки, описывающей нашу задачу, как поиск тотального порядка, для которого заданный порядок является подмножеством.

Ациклические отношения имеют топологическую сортировку.

Отсутствие циклов является очевидным необходимым условием существования топологической сортировки (тотального порядка, включающего исходное отношение). Но достаточно ли этого? Если мы имеем ациклическое отношение, можем ли мы всегда произвести топологическую сортировку – полный порядок, включающий отношение?

Справедлива следующая теорема.

Теорема о топологической сортировке

Для любого ациклического отношения r над конечным множеством *A* существует тотальное отношение порядка *t* над *A*, такое, что $r \subseteq t$.

Для доказательства этой теоремы можно было бы использовать то наблюдение, что $r \subseteq r^+$, где r^+ – это транзитивное замыкание *r*, и предыдущее доказательство того, что r^+ является отношением порядка. Этого достаточно, чтобы расширить r^+ до *полного* отношения порядка. Но для наших целей более интересно другое доказательство. Это *конструктивное* доказательство (основанное на теореме об отсутствии предшественника), позволяющее нам непосредственно получить схему алгоритма.

Доказательство ведется индукцией по числу элементов *n* множества *A*. Если $n = 0$, множество пусто, единственно возможным отношением является пустое отношение (пустое множество пар элементов из *A*), которое является полным порядком. Это доказывает базис индукции.

Если вы предпочитаете иное, в качестве базиса можно выбрать случай $n = 1$, для которого *A* состоит из одного элемента *x*. Хотя теперь *A* не пусто, единственное ациклическое отношение в *A* снова является пустым отношением, так как единственная пара [*x*, *x*], которую можно создать, образует цикл.

Для индукционного шага предположим, что теорема выполняется над множествами из *n* элементов, и рассмотрим ациклическое отношение на множествах из *n* + 1 элементов. Рисунок поясняет идею доказательства:

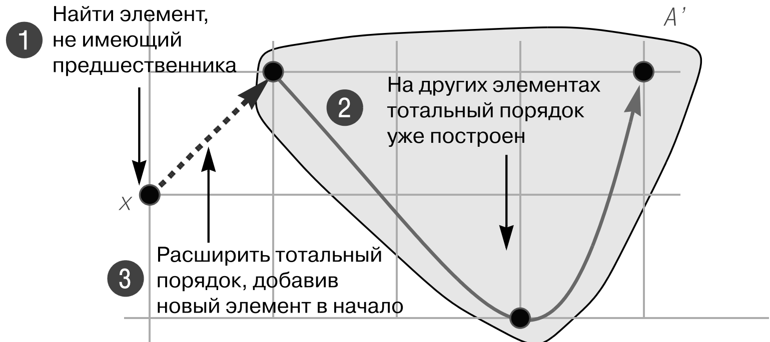


Рис. 15.9. Расширение, дополняющее топологическую сортировку

Теорема об отсутствии предшественника говорит нам, что A имеет по крайней мере один элемент, не имеющий предшественника. Пусть x – такой элемент. Пусть A' – множество элементов A за исключением x. Пусть r' – отношение на A', содержащее все пары из r, за исключением тех, что содержат x. Очевидно, что r' является ациклическим отношением на A'. По предположению индукции, так как A' имеет n элементов, существует полный порядок t' над A', совместимый с r' (это то же самое, что сказать $r' \subseteq t'$). Теперь рассмотрим отношение t над A, состоящее из следующих пар:

- все пары в t';
- все пары вида $[x, y]$, где y – элемент из A'.

> Если вы предпочитаете думать о полном порядке как о перечислении, то можно рассматривать t как перечисление элементов A, *которое* начинается с x и продолжается перечислением A', заданным t'.

Нетрудно видеть, что t – это полный порядок, и что $r \subseteq t$; это дает нам полный порядок, совместимый с r, что и доказывает теорему.

Теорема о топологической сортировке является математическим обоснованием программы, которую мы собираемся построить. Более того, доказательство непосредственно приводит к основной идее алгоритма.

15.3. Практические соображения

С теоретическими основами все ясно, так что можно приступать к поиску инженерного решения. Ядром является алгоритм топологической сортировки, но прежде следует проанализировать ограничения производительности и определить рамки инженерии программы.

Требования производительности

Что можно сказать об ожидаемой емкостной и временной сложности алгоритма?

Входом для алгоритма является множество элементов и множество ограничений. Обозначим через n число элементов, m – число ограничений.

Алгоритм должен (в случае ациклического отношения) выполнить:

- по меньшей мере одну операцию для каждого ограничения (так как игнорирование любого из ограничений могло бы привести к ошибочному порядку построения вывода);
- по меньшей мере одну операцию для каждого элемента, хотя бы для того, чтобы добавить элемент в вывод.

Так что наилучшее время, на которое можно надеяться, – это $O(n + m)$.

Самое удивительное, что разрабатываемый ниже алгоритм топологической сортировки достигает этой нижней теоретической оценки, как по времени, так и по памяти.

Каркас класса

Чисто алгоритмическое решение могло бы использовать функцию в форме:

```
topologically_sorted (elements:…; constraints:…): LIST[…]
        -- Перечисление элементов множества elements
        -- в порядке, совместимом с множеством ограничений constraints.
```

Предполагается, что входные типы позволяют задать множества элементов и ограничений – *elements* и *costraints,* соответствующие нашим прежним обозначениям *A* и *r*.

Лучшее решение – как покажет дальнейшая разработка – дает ОО-подход с построением класса *TOPOLOGICAL_SORTED*, любой экземпляр которого представляет задачу топологической сортировки. Структуры данных, представляющие элементы и ограничения, будут атрибутами класса, формируемые процедурами инициализации, такими как *record_element* и *record_constraint*.

Вместо функции *topologically_sorted*, как выше, будем иметь в классе:

- процедуру *process*, выполняющую процесс топологической сортировки;
- запрос *sorted*, который возвращает список элементов, созданный при работе процедуры *process*.

Этот каркас даст нам больше гибкости, и мы получим возможность дополнить его многими полезными свойствами.

Вход и выход

Множества элементов *A* и ограничений *r* могут поступать из различных источников. Например, мы могли бы иметь файл, перечисляющий ограничения, каждое из которых задается одной строкой файла:

```
Map Louvre
Map Orsay
Pass Louvre
Pass Orsay
Money Pass
```

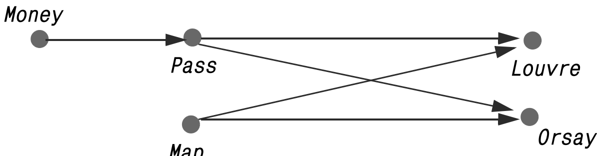


Рис. 15.10. Элементы и ограничения

Возможно, было бы полезно иметь отдельный файл, перечисляющий все элементы или, по меньшей мере, элементы, не включенные ни в одно из ограничений (мы не можем догадаться о существовании таких элементов, анализируя ограничения, но они должны быть частью вывода).

В других условиях ввод мог бы осуществляться интерактивно, используя программу или Web-форму. В примерах, таких как упорядочение прямоугольников, отображаемых на экране, упорядочение терминов словаря или методов класса, формат данных будет различным.

Для обеспечения общности сделаем наш базовый класс универсальным: *TOPOLOGICAL_SORTED*[*G*], где параметр *G* задает тип элементов. Тогда результат запроса *sorted*, который обозначает топологически отсортированный список элементов, созданный в *process*, имеет тип *LIST*[*G*]. Две процедуры инициализации, о которых говорилось выше, имеют сигнатуры:

```
record_element (e: G)
record_constraint (e, f: G)
```

Полная форма алгоритма

Рассмотрим ациклическое отношение *r* на множестве элементов *A*. Так как нам необходимы имена, используемые в программе, то позвольте полагать, не предрешая выбор реализации, что класс *TOPOLOGICAL_SORTED* имеет доступные запросы *elements* и *constraints*. Общая схема для алгоритма топологической сортировки в процедуре *process* такова:

```
from …until elements.is_empty loop
    "Пусть x - элемент без предшественников в ограничениях"
    "Рассматривать x как следующий элемент sorted"
    "Удалить x из множества элементов"
    "Удалить все пары, начинающиеся x, из множества ограничений)"
end
```

Уточненная нужным образом эта форма будет работать при условии, что мы начинаем с ациклического отношения (или в специальном случае – с отношения порядка).

Четыре предложения псевдокода, показанные выше, будут повторяться в оставшейся части главы.

Теоремы о топологической сортировке и об отсутствии предшественника дают нам обоснование решения, но его еще нужно выразить в идее инварианта цикла и варианта (надеюсь, цикл без инварианта – для вас ужасная картина, не правда ли?)

```
process
      — Выполняет в sorted перечисление элементов множества elements
      — в порядке, совместимом с множеством ограничений constraints.
    require
            — "constraints описывают ациклическое отношение"
    do
      from
            create {….} sorted. make
      invariant
            — "constraints описывают ациклическое отношение"
      until
            elements. is_empty
      loop — Как и ранее, за тем исключением, что явно используется результат sorted:
            "Пусть x - элемент без предшественников в ограничениях"
            sorted. extend (x)
            "Удалить x из множества элементов"
            "Удалить все пары, начинающиеся x, из множества ограничений"
      variant
            elements. count
      end
```

```
  ensure
  – "sorted представляет топологическую сортировку elements, согласованную
  – с constraints"
end
```

Циклы в ограничениях

Версия схемы алгоритма, в принципе, корректна, но не подходит для большинства приложений, встречающихся в реальной жизни.

Проблема в предусловии, требующем, чтобы на вход было подано ациклическое отношение. Программа топологической сортировки получает вход в виде индивидуально упорядоченных пар, например, [*Map, Louvre*] или [*Map, Orsey*], как выше. Такой вход может готовиться человеком, а ему свойственно ошибаться, и он имеет право на ошибку, что недопустимо в программе (при рассмотрении сборки промышленных изделий могут встречаться тысячи узлов и десятки тысяч ограничений между ними, задающих порядок сборки).

В примере с терминами словаря мы не можем надеяться, что не встретятся два или более термина, ссылающихся друг на друга в своих определениях (следовательно, создающих циклы), но у нас нет способа принуждать авторов словаря. Справедливо обратное – автор может требовать от программы: «Упорядочьте в словаре термины так, чтобы определение термина встречалось раньше его использования. Но, кстати, если вы обнаружите взаимные ссылки терминов, сообщите мне о них, чтобы я мог скорректировать определения».

Аналогично задача упорядочения методов класса, чтобы определение метода предшествовало его вызову, невозможна в случае косвенной рекурсии, хотя это и не является ошибкой. Так что интерес может представлять применение топологической сортировки к нециклической части графа вызовов и получение отчета о любых оставшихся циклах.

Эти рассмотрения приводят к изменениям контракта метода *process*, более благосклонного к своим клиентам. Прежний контракт:

```
require
                  – "constraints описывают ациклическое отношение"
ensure
                  – "sorted представляет топологическую сортировку elements,
                  – согласованную с constraints"
```

заменяется новым, более реалистичным контрактом:

```
– (Нет предусловия)
ensure
          – "sorted представляет топологическую сортировку, согласованную
          – с constraints, для всех членов elements, не входящих в цикл"
```

Этого все же недостаточно. Класс должен предоставлять своим клиентам отчет о возникающих циклах и указывать, какие элементы в них включены. Одним из способов обеспечения такой функциональности является введение булевской функции.

```
has_cycle: BOOLEAN
          – Содержат ли циклы отношение, заданное elements и constraints?
  do …end
```

Говоря неформально, нужна функция, которая будет возвращать список элементов, включенных в цикл (void, если и только если *has_cycle* имеет значение false). Концептуально это звучит разумно, но это не лучший способ, так как вычислительно представляет дорогое решение. Нахождение циклов – задача функции *has_cycle*, фактически является столь же трудной по времени и памяти, как и сама задача топологической сортировки, но если мы попытаемся выполнять топологическую сортировку без предусловия, то без особых затрат можем обнаружить циклы в процессе работы.

Теорема об отсутствии предшественника подсказывает нам, что цикл можно найти как побочный бонус процесса топологической сортировки.

- Как следует из приведенного выше цикла, на каждом шаге разыскивается элемент, не имеющий предшественника, и это делается до тех пор, пока множество не станет пустым.
- Теорема говорит нам, что для ацикличного отношения процесс завершится, то есть на каждом шаге непустого множества такой элемент существует.
- Если мы не можем найти элемент, не имеющий предшественника, а множество еще не пусто, то это означает, что оставшиеся элементы составляют, по меньшей мере, один цикл. Мы можем завершить алгоритм, отчитавшись, что полная топологическая сортировка невозможна. Это благоприятная форма завершения, так как мы топологически отсортировали элементы, не создающие цикл, и имеем возможность сказать клиенту, какие именно элементы и ограничения создают проблему, – оставшиеся после сортировки.

Рассматриваемая далее программа топологической сортировки не имеет предусловия, а ее инвариант цикла также изменен.

Вместо прежнего инварианта:

```
–"constraints описывают ациклическое отношение на elements"
```

будем теперь использовать его ослабленную форму:

```
–"constraints описывают подмножество исходного отношения на elements"
```

Отсюда следует, что

```
–"Любой цикл в constraints присутствует в исходном отношении"
```

а также:

```
–"constraints описывает ациклическое отношение, если исходное  отношение ациклично".
```

Это та основа, которой мы будем придерживаться. Как следствие, мы больше не будем использовать в качестве условия выхода из цикла *elements.is_empty*, как ранее, так как непустое множество *elements* не гарантирует больше, что можно корректно выполнить оператор:

```
"Пусть x – элемент без предшественников в ограничениях"
```

В качестве нового условия выхода будем иметь:

```
"Нет элементов без предшественников в ограничениях"
```

Отрицание этого условия означает, что есть по крайней мере один элемент без предшественников, – а это гарантия того, что в теле цикла можно найти очередного кандидата, включаемого в результат выхода.

Полная организация класса

Мы можем теперь определить полную форму класса, который будет служить каркасом нашего решения:

```
class
    TOPOLOGICAL_SORTER [G -> HASHABLE]
feature {NONE} – Внутренние структуры данных
              …Смотри следующие разделы этой главы…
feature – Инициализация
    record_element (a: G)
              – Включить a в множество элементов.
      require
              not_sorted: not done
      do
              …Смотри следующие разделы…
      end
    record_constraint (a, b: G)
      – Включить [a, b] в ограничения.
      require
              not_sorted: not done
      do
              …Смотри следующие разделы…
      end
feature – Status report
    done: BOOLEAN
              – Выполнена ли топологическая сортировка?
feature – Element change
    process
              – Выполнить топологическую сортировку над всеми применимыми элементами.
              – Результаты доступны через sorted, cycle-found и cyclists.
      require
              not_sorted: not done
      do
              …Смотри следующие разделы…
      ensure
              sorted: done
      end
feature – Access
    cycle_found: BOOLEAN
              – Исходное ограничение приводит ли к циклу?
    cyclists: LIST [G]
              – Элементы, включенные в какой-либо цикл.
    sorted: LIST [G]
              – Список из всех элементов, которые допускают упорядочивание,
```

```
            — согласованное с ограничениями
feature — Status setting
    reset
            — Допустить дальнейшее обновление элементов и ограничений.
            do
                    done:= False
                    cycle_found:= False; cyclists:= Void; processed_count:= 0
            ensure
                    fresh: not done
            end
invariant
    elements_exist: elements /= Void
    constraints_exist: constraints /= Void
    cyclists_only_if_cycle: done implies (cycle_found = (cyclists /= Void))
end
```

Компоненты класса были перечислены в порядке, облегчающем последовательное чтение. Порядок, рекомендуемый стандартом, будет восстановлен в окончательной версии класса.

Класс является универсальным, его родовой параметр *G* представляет тип элементов. Запись *G* -> *HASHTABLE* означает, что мы «ограничиваем» *G* (понятие, вводимое в следующей главе) типом *HASHTABLE.* Причина ограничения типа прояснится немного позже.

Алгоритм будет основываться на внутренних структурах данных, проектируемых в следующих разделах этой главы. Поскольку соответствующие методы создаются для внутреннего использования, они не будут доступны клиентам класса и потому объявлены как **feature** {*NONE*}.

Сразу же после того, как метод *process* выполнит свою работу, его результаты будут доступны клиентам через несколько связанных запросов.

- Булевский запрос *done*, позволяющий клиентам выяснить, выполнена ли топологическая сортировка. При инициализации он получает значение false.
- Список, формируемый запросом *sorted*, дает порядок, совместимый со всеми ограничениями, и включает все элементы, не входящие в цикл.

Булевский запрос *cycle_found* показывает, что есть элементы, включенные в какой-либо цикл.

- Список всех таких элементов, включенных в цикл, представлен запросом *cyclists*. Предложение инварианта *cyclists_only_if_cycle* говорит нам, что запрос имеет смысл только при условии истинности *cycle_found*.

Клиент, желающий выполнить топологическую сортировку, обычно будет использовать наш класс следующим образом:

```
your_structure: TOPOLOGICAL_SORTER [YOUR_ELEMENT_TYPE]
…
create your_structure
…Вызовы вида your_structure.record_element(x) для записи элементов …
…и your_structure.record_constraint(x, y) для записи ограничений…
your_structure. process
— Теперь становятся доступными результаты топологической сортировки
```

```
— your_structure.sorted
if your_structure. cycle_found then
    — Теперь становятся доступными элементы, включенные в цикл
    — your_structure.cyclists …
end
```

Было бы желательно для согласованности поставлять запросы *sorted*, *cycle_found* и *cyclists* с предусловием *done*, но мы временно будем опускать предусловие.

Однако предусловие **not** *done* задано для процедур инициализации *record_element* и *record_constraint*, так же как и для процедуры топологической сортировки *process,* имеющей постусловие *done*. Как результат, попытка повторного вызова процедуры *process* на одном и том же экземпляре класса будет приводить к ошибке. Конечно, запросы можно вызывать сколь угодно много раз. Процедура *reset* добавлена на случай, когда вы хотите явно добавлять элементы и ограничения уже после вызова *process* для подготовки нового его вызова.

Процедура *reset* просто устанавливает *done* равным false, не производя очистки предыдущих элементов и ограничений. Мы можем добавить процедуру *forget*, которая вызовет *reset* и очистит структуры данных. Но в этом случае клиенту проще создать новый экземпляр *TOPOLOGICAL_SORTER*.

15.4. Базисный алгоритм

Теперь мы можем приступить к полной реализации ключевой части нашего решения – процедуре *process*.

Цикл

Общий алгоритм этого метода уже написан, осталось адаптировать его с учетом всех сделанных уточнений (ослабление инварианта, использование метода *sorted*, представляющего результат в *TOPOLOGICAL_SORTER*). В результате получаем:

```
from
            create {…} sorted. make
until
     "Нет элемента без предшественников"
Loop
     "Пусть x — элемент без предшественников"
            sorted. extend (x)
     "Удалить x из множества элементов"
     Удалить все ограничения, начинающиеся с x
end
if "остался хоть один элемент" then — Отчет о цикле:
     cycle_found:= True
     "Вставьте эти элементы в cyclists"
end
```

Все, что осталось, – но не спешите слишком радоваться, главные трудности еще впереди, – это уточнить псевдокод, превратив его в код программный. Заключительная часть (отчет о

циклах) будет прямым следствием всего остального. Главная задача состоит в уточнении четырех операторов псевдокода, точнее, трех, поскольку мы можем рассматривать первые два как одну операцию. Сосредоточимся на этих операциях в процессе поиска эффективного алгоритма.

Топологическая сортировка: базисные операции

T1 Найти элемент без предшественников – или отчитаться об отсутствии такового.
T2 Для заданного элемента *x* удалить его из множества элементов.
T3 Для заданного элемента *x* удалить из множества ограничений все ограничения, начинающиеся этим элементом (все пары вида [*x, y*]).

Мы должны найти такие представления данных (множеств элементов и ограничений), которые делают эти операции эффективными, насколько это возможно. Структуры данных будут появляться как закрытые компоненты класса в разделах, резервированных для этих целей.

Приведенная выше схема класса оставила незаполненными тела двух методов класса: *add_element* и *add_constraint*. Нам предстоит дополнить методы, основываясь на структурах данных, которые мы сейчас спроектируем.

«Естественный» выбор структур данных

Для нашей первой попытки построения структуры данных естественно выбрать представление, которое непосредственно моделирует входные данные в том виде, как они к нам приходят (наш небольшой опыт в изучении программирования говорит, что не следует утверждать, что некоторое решение является «естественным». То, что кажется естественным мне, вовсе может не казаться таковым для вас, а то, что естественно для меня и для вас, может оказаться довольно посредственным решением).

Чаще всего входные данные могли бы поступать в виде списка элементов и списка ограничений. Мы могли бы использовать атрибуты класса, которые непосредственно отображают эти структуры данных (сделав их закрытыми для клиентов, размещая их в секции **feature** {*NONE*}):

```
elements: LINKED_LIST [G]
constraints: LINKED_LIST [TUPLE [G, G]]
```

В нашем примере структуры данных будут выглядеть подобным образом:

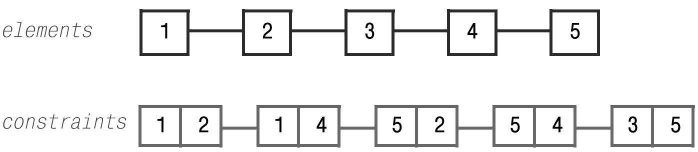


Рис. 15.11. Элементы и ограничения

Для простоты предполагается, что мы присвоили номера нашим элементам следующим образом (рис. 15.12).

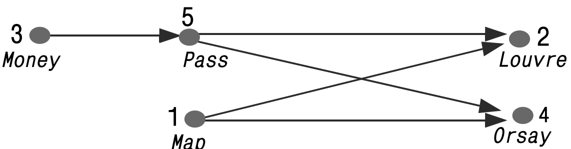


Рис. 15.12. Нумерация элементов

Анализ производительности естественного решения

Можем ли мы реализовать при таком представлении данных операции T1, T2, T3 и процедуры *record_element* и *record_constaint*, и если да, то какова цена решения – сложность по времени и по памяти?

Две процедуры реализуются непосредственно. Например, для выполнения *record_constaint* достаточно вызывать стандартный метод:

```
constraints. extend ([x, y])
```

Новая пара [*x,y*] добавится в конец списка ограничений. Подобным образом *record_element*(*x*) реализуется вызовом *elements.extend*(*x*).

> При применении этих операторов, как обычно, следует быть уверенным, что объекты *elements* и *constaints* имеют значения, отличные от void. Соответствующие операторы создания *create* могут появляться в форме *default_create* или вызываться по требованию, когда в них возникает необходимость. Все это справедливо и для других структур данных, рассматриваемых ниже.

Это представление можно использовать и при реализации других операций. Давайте проанализируем их стоимость, когда у нас есть *n* элементов и *m* ограничений.

- Для поиска элемента без предшественников (T1) можно обойти список ограничений *constaints* и считать предшественников для каждого элемента, затем обойти список элементов для поиска тех, у кого число предшественников равно нулю. Первый обход потребует $O(m)$ операций, второй – $O(n)$, и поскольку это нужно делать на каждом шаге, в целом понадобится операций $O(n*\mathrm{m} + n^2)$.
- Удаление элемента (T2) требует на каждом шаге $O(n)$ операций, а в целом – $O(n^2)$.
- Удаление ограничений (T3) требует на каждом шаге O(m) операций, а в целом – $O(m * n)$. Предполагается, что мы имеем единый список ограничений, а потому для удаления элементов, начинающихся с некоторого заданного *x*, потребуется обход всего списка.

В практических приложениях следует ожидать, что для каждого элемента задается несколько ограничений, так что $m > n$, а потому $O(m * n)$ хуже, чем $O(n^2)$. Поскольку в реальных задачах *n* может быть велико, сложность $O(n^2)$ представляется неприемлемо высокой.

Так что наш первый выбор структуры данных дает решение задачи, но его производительность не позволяет построить масштабируемое приложение, пригодное для задач большой размерности.

Дублирование информации

К счастью, мы можем найти решение лучшее, чем «естественное». Как показывает опыт, мы не всегда должны использовать структуры данных в том виде, как они к нам приходят. Списки *elements* и *constraints* – это непосредственное отражение данных внешнего мира, естественного представления данных человеком, который вводит задачи и описывает ограничения между задачами. Но форма данных, простая и естественная для внешнего мира, вовсе не является лучшей формой для алгоритма, предназначенного для обработки этих данных некоторым специальным образом. Вместо того чтобы слепо следовать форме приходящих данных, алгоритм может начинаться с этапа инициализации, который преобразует данные в формат, наилучшим образом приспособленный к их обработке.

Следующие структуры данных помогают выполнить работу топологической сортировки – задачи T1-T3 – удобно и быстро:

```
successors: ARRAY [LINKED_LIST [INTEGER]]
            -- Индексами массива являются номера элементов. Для каждого элемента x
            -- задается список его непосредственных последователей - элементов y,
            -- таких, что есть ограничение [x, y].
predecessor_count: ARRAY [INTEGER]
            -- Индексами массива являются номера элементов.
            -- Для каждого элемента x указывается число его непосредственных
            -- предшественников.
```

Следующий рисунок иллюстрирует представление данных для нашего примера.

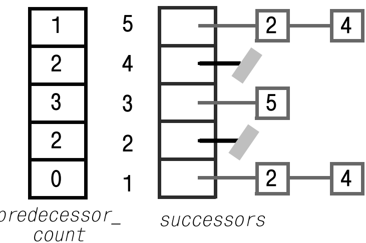


Рис. 15.13. Число предшественников и список последователей

На следующем рисунке приведена уже хорошо нам знакомая схема, где введена нумерация элементов. Она поясняет предыдущий рисунок. По схеме ясно, что, например, у задачи 1 предшественников нет, но есть два последователя, а у задачи 5 – один предшественник и те же два последователя (2 и 4).

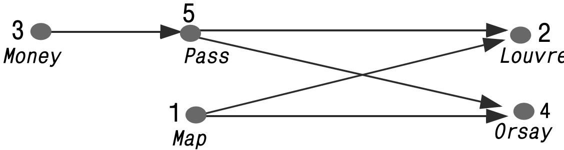


Рис. 15.14. Знакомая схема с нумерацией элементов

В этом представлении интересно то, что массив, задающий число предшественников, является избыточным, поскольку всю хранимую в нем информацию можно извлечь из массива последователей. Но ничего ошибочного в таком представлении информации нет, поскольку, как мы увидим, за счет этого дублирования можно добиться существенного улучшения времени вычислений.

Нахождение компромисса **память-время** является ключевой составляющей проектирования хорошего алгоритма. Конечно, компромисс должен быть приемлемым. В данном случае наша цель — получить алгоритм с временной сложностью $O(m + n)$, сохраняя ту же сложность и для памяти, требуемой для хранения данных. Такая память нужна для хранения массива *successors*, добавление массива *predecessor_count* с емкостью $O(n)$ не меняет емкостной сложности алгоритма в целом.

Исходные структуры данных – *elements* и *constraints* – также требуют памяти $O(m + n)$.

Украсим алгоритм инвариантом класса

Удобно для пущей ясности добавить запрос, доступный клиентам:

```
count: INTEGER
            -- Число элементов
```

Также полезно для улучшения читабельности добавить инварианты класса. Два последних предложения выражены неформально:

```
elements.count = count
predecessor_count. count = count
successors.count = count
-- Для каждого i из интервала 1..count элемент predecessor_count[i]
-- задает число предшественников i-го элемента для заданных ограничений.
-- Для каждого i из интервала 1..count элемент successors[i]
-- задает список всех последователей i-го элемента для заданных ограничений
-- или void, если последователей нет.
```

Нумерация элементов

Чтобы использовать массив, нужно некоторым образом пронумеровать наши элементы. Ранее это было сделано в интересах удобства обращения. Теперь это становится необходимостью, связанной с нашим выбором структуры данных.

Значит ли это, что теперь требуется изменить родовой параметр класса *TOPOLOGICAL_SORTER*[*G*], поскольку все манипуляции над элементами будут теперь использовать их целочисленные номера? Абсолютно нет. Остается необходимым для выразительности создать механизм, применимый к элементам любого типа. На практике для этого потребуется хэш-таблица и массив:

```
index_of_element: HASH_TABLE [INTEGER, G]
            -- Для каждого элемента дает его индекс
element_of_index: ARRAY [G]
            -- Для каждого индекса дает ассоциированный с ним элемент
```

Элемент хэш-таблицы *index_of_element*[*e*] дает целое *x* – индекс элемента *e* типа *G*. В свою очередь, *element_of_index*[*x*] = *e*.

В обоих случаях используется нотация с квадратными скобками.

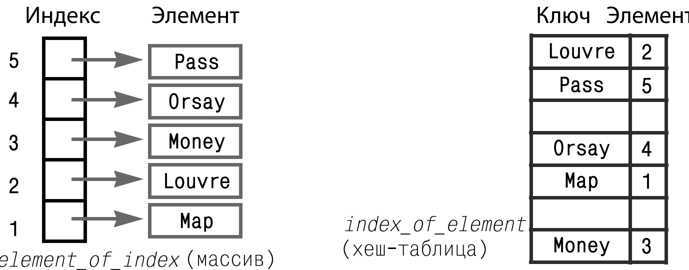


Рис. 15.15. Взаимное отображение элементов и индексов

Обе эти структуры при разумной реализации требуют памяти O(*n*).

Для определения хэш-таблицы с элементами типа *G* требуется, чтобы тип *G* удовлетворял ограничениям наследования – был потомком класса *HASHTABLE* и объявлен как *G* -> *HASHTABLE* (ограниченная универсальность будет подробно рассматриваться в следующей главе, сейчас же понятно, что элементы должны допускать построение хэш-функции).

Класс *TOPOLOGICAL_SORTER*[*G* -> *HASHTABLE*] не будет экспортировать компоненты *index_of_element* и *element_of_index*, так как они необходимы только для целей реализации, но мы должны позволять клиентам находить элементы, поскольку это является частью общей задачи работы с элементами, поэтому мы будем экспортировать запрос:

```
has_element (e: G): BOOLEAN
            -- Является ли e одним из элементов топологической сортировки?
    do
            Result:= index_of_element. has (e)
    ensure
            consistent: Result = index_of_element.has (e) and then
                    index_of_element [e] >= 1 and then
                    index_of_element [e] <= element_of_index. count and then
                    element_of_index [index_of_element [e]] = e
end
```

Убедитесь, что вы понимаете постусловие.

Давайте теперь докажем, что новые структуры данных позволяют добиться нашей цели – времени O(*m* + *n*). Необходимо рассмотреть два аспекта: выполнение операций T1-T3 и затраты на инициализацию – создание структур данных *predecessor_count* и *successors*. Оба аспекта важны. Если бы для новых структур данных операции T1-T3 выполнялись бы за время O(*m* + *n*), но для создания структур требовалось бы O(*m* * *n*) времени, то в целом никакого выигрыша мы бы не получили.

Представляется, что несложно реализовать построение требуемых структур данных за время O(*m* + *n*). Обрабатывая последовательно каждое ограничение [*x, y*], зная *y*, можно уве-

личить на 1 значение соответствующего элемента *predecessor_count*, а зная *x*, можно добавить последователя *y* в список соответствующего элемента *successors*. Обе операции выполняются за константное время. Опуская детали, будем считать далее, что построение структур данных за время O($m + n$) возможно и сосредоточимся на рассмотрении операций T1-T3.

Базисные операции

Начнем с T3: «Для заданного элемента *x* удалить из множества ограничений все ограничения, начинающиеся этим элементом (все пары вида [*x, y*])». Если мы знаем номер *x*, то выполнить удаление совсем просто.

L1 Нам больше не нужен список последователей элемента *x*. Мы можем очистить этот список простым присваиванием *successors*[*x*] = Void. На практике и присваивание не нужно, так как алгоритм никогда не посетит вход *x* этого массива. Но даже если и выполнять эту операцию, то она выполняется за время O(1) для одного элемента и O(n) для всех элементов. Прекрасно!

L2 Нам требуется также обновить соответствующие элементы в массиве *predecessor_count*. Для каждого последователя *y* элемента *x* нужно уменьшить *predecessor_count*[*y*] на 1, так как в связи с удалением ограничения уменьшается число предшественников, и для выполнения операции потребуется обход списка, связанного с элементом *x*. Это будет делаться непосредственно в цикле, чей код появится ниже. Процесс будет выполняться самое большее один раз за всю обработку для каждого ограничения, так что он потребует времени O(m). Снова прекрасно!

В целом операция T3 требует времени O($m + n$) в худшем случае.

Вся только что описанная обработка сопровождается инвариантами класса, выражающими тот факт, что массивы *predecessor_count* и *successors* в полной мере отражают структуру оставшихся ограничений отношения.

Рассмотрим теперь операцию T2: «Для заданного элемента x удалить его из множества элементов». Фактически, для нашей новой структуры данных нам ничего не нужно делать. Выполняя T3, мы уже позаботились обо всем, что нужно было сделать, удаляя ограничение, начинающееся с *x*. Отлично!

Осталась операция T1: «Найти элемент без предшественников – или уведомить об отсутствии такового». Для этого достаточно выполнить обход массива *predecessor_count* и найти элемент со значением 0. Но для этого понадобится время O(n), а для всех элементов O(n^2). Это плохо!

Нам недостает еще одной структуры данных!

Кандидаты

Нам не избежать O(n) обхода массива *predecessor_count* при инициализации (предложения **from** нашего главного цикла) для поиска первоначальных кандидатов – элементов без предшественников в исходном отношении. Если только не каждый элемент включен в цикл, то такие элементы найдутся. Это потребует O(n) времени, но выполнить эту операцию придется лишь один раз, так что пока все хорошо. По ходу процесса топологической сортировки обнаружение кандидатов на удаление – элементов без предшественников – можно получить в качестве побочного эффекта выполняемой операции T3. Действительно, на этапе L2 мы уменьшаем на 1 число предшественников. Если при этом это число становится равным нулю, то найден новый кандидат. Если ранее этап L2 можно было бы записать в виде:

```
        — Уменьшить на единицу число предшественников y:
  predecessor_count [y]:= predecessor_count [y] - 1
```

то теперь соответствующий код будет выглядеть так:

```
    — Уменьшить на единицу число предшественников y
    — и проверить, не становится ли y кандидатом:
  predecessor_count [y]:= predecessor_count [y] - 1
            [3]
  if predecessor_count [y] = 0 then
            "Записать, что y не имеет предшественников"
  end
```

«Записать, что *y* не имеет предшественников» может быть реализовано как добавление в структуру *candidates*, которая будет заполняться при инициализации и пополняться по ходу обработки. Ее элементами являются еще не обработанные элементы, не имеющие предшественников. Какую конкретную структуру следует выбрать для *candidates?* Для алгоритма топологической сортировки точный выбор не имеет значения. Важно лишь, чтобы эта структура поддерживала следующие 5 операций.

```
feature — Access
  item: G
                  — Получить ранее вставленный элемент.
    require
                  not_empty: not is_empty
feature — Measurement
  count: INTEGER
                  — Число элементов.
    ensure
                  non_negative: Result >= 0
feature — Status report
  is_empty: BOOLEAN
            — Пуста ли структура?
    ensure
            definition: Result = (count = 0)
feature -- Element change
  put (x: G)
                  — Вставить элемент x.
    ensure
                  one_more: count = old count + 1
  remove: G
                  — Удалить прочитанный элемент.
    require
                  not_empty: not is_empty
    ensure
                  one_fewer: count = old count - 1
```

Структуры данных с такими свойствами называются **распределителями**.

Рис. 15.16. Распределитель

Как вы помните, основная идея распределителя в том, что не вы задаете, какой элемент будет получен и удален из распределителя, – стратегия задается типом распределителя. Стеки характеризуются политикой LIFO, очереди – FIFO.

Для топологической сортировки любой распределитель будет выполнять нужную работу. Выбор определенного вида влияет на фактический порядок, в котором будут выводиться элементы (решений, совместимых с ограничениями, может быть несколько). Так что это тот рычаг, которым можно управлять, оптимизируя результат по некоторому критерию. Он позволяет рассматривать семейство алгоритмов топологической сортировки.

Мы можем рассматривать распределитель кандидатов в виде:

```
candidates: PRIORITY_QUEUE [INTEGER]
                  – Элементы без предшественников, готовые к удалению
                  – Дополнительное предложение для инварианта:
                  – predecessor_count[x] = 0 для каждого элемента x из массива candidates
```

Реализация распределителя стеком или очередью (*STACK* или *QUEUE*) допустима, но очередь с приоритетами является более общим видом, где каждый элемент может сопровождаться приоритетом. В такой очереди элементы отсортированы по приоритетам, и операции *item* и *remove* выполняются для элемента с наибольшим приоритетом. Стек и очередь – это специальные случаи, когда приоритет задается порядком поступления элементов. Распределитель *PRIORITY_QUEUE* позволяет вам, играя приоритетами, управлять политикой выбора.

Цикл, заключительный вид

Мы можем теперь выписать главный цикл алгоритма топологической сортировки – тело процедуры *process* со всеми деталями. Операторы псевдокода для наглядности сохраним и в этой версии как комментарии. Процедура должна объявить локальные переменные *x* и *y* типа *INTEGER* и *x_successors* типа *LIST*[*INTEGER*], сохраняющего последователей конкретного элемента. Мы также добавим целочисленную переменную *processed_count*, используемую далее, чтобы сохранять историю того, как много элементов уже обработано.

```
from
    create sorted. make
    find_initial_candidates                     – Смотри далее
invariant
    – “Структуры данных представляют подмножество исходных элементов
    – и соответствующее подмножество исходного отношения”
until
    candidates.is_empty
loop
                    – “Пусть x - элемент без предшественников в ограничениях”
    x:= candidates.item; candidates.remove
    sorted.extend (element_of_index [x])
                    – “Удалим x и все пары в ограничениях, начинающиеся с x”
    x_successors:= successors [x] – список
    from x_successors.start until x_successors. after loop
            y:= x_successors. item
                            – Следующие несколько строчек взяты из [3]:
            predecessor_count [ y]:= predecessor_count [y] - 1
            if predecessor_count [ y] = 0 then
                            – “Записать, что теперь y не имеет предшественников”
                    candidates. put (y)
            end
            x_successors. forth end
            processed_count:= processed_count + 1
variant
            count - processed_count
end
report_cycles                   – Смотри далее
done:= True
```

Этот алгоритм предполагает, что массивы *predecessor_count* и *successors* правильно сформированы, как и должно быть перед любым вызовом *process*. Детали инициализации появятся чуть позже.

Процедура *find_initial_candidates* должна наполнить распределитель *candidates* элементами, изначально не имеющими предшественников. Она реализуется просто:

```
find_initial_candidates
            – Поместите в массив элементы без предшественников.
    local
            x: INTEGER
    do
      if candidates = Void then create candidates end
      from x:= 1 until x > count loop
            if predecessor_count [x] = 0 then
                    candidates. put (x)
```

```
			end
			x:= x + 1
		end
	end
```

Это обход за время O(n). Без введения такого массива пришлось бы выполнять такой обход на каждом шаге цикла. Теперь достаточно выполнить его один раз в самом начале работы.

Не является ошибкой, если в процедуре не будут найдены элементы, удовлетворяющие условию *predecessor_count*[*x*] = 0. Это просто означает, что структура *candidates* пуста, что цикл завершится незамедлительно и что все элементы включены в какой-либо цикл.

Процедура *process* по завершении цикла должна выполнить еще одну важную часть работы – она должна уведомить клиента о циклах, встречающихся в отношении. Это и делается при вызове процедуры *report_cycles*. Чтобы ее реализовать, предварительно заметим, что цикл завершается, когда в массиве *candidates* не остается элементов. Если исходное отношение было ациклическим, то будут обработаны все элементы, так что можно использовать введенную ранее переменную *processed_count*, чтобы понять, остались ли элементы, и, если да, то сколько их:

```
report_cycles
			– Сделать информацию о циклах доступной клиентам.
	do
			if processed_count < count then
						– В исходном отношении есть цикл!!
				cycle_found:= True
				create {LINKED_LIST [G]} cyclists. make
				from x:= 1 until x > count loop
					if predecessor_count [x] /= 0 then
						– x включен в цикл
					cyclists. extend (element_of_index [x])
					x:= x + 1
				end
			end
	end
```

Инициализация и время ее выполнения

Мы достигли эффективной реализации за время O($m + n$) ядра топологической сортировки – ее основного цикла. Этому способствовали три структуры данных, специально спроектированные для этих целей – массивы *predecessor_count* и *successors* и распределитель *candidates*. Дополняя работу, следует убедиться, что инициализация не нарушает требуемых ограничений на время работы.

Инициализация должна выполнять:

- *record_element*(*e*) для каждого элемента – всего n раз;
- *record_constraint*(*e*, *f*) для каждого ограничения – всего m раз.

Работа *record_element*(*e*) состоит в том, чтобы присвоить номер элементу *e*, так чтобы в дальнейшей работе можно было бы использовать целые, а не сами элементы, имеющие тип *G*.

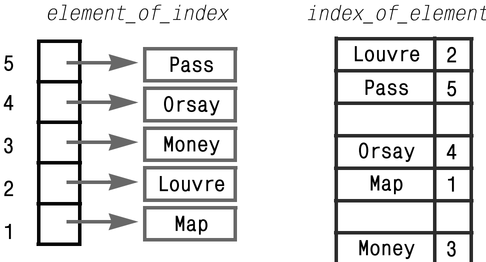


Рис. 15.17. Массив и хеш-таблица

Это делается согласованным заполнением массива *element_of_index* и хэш-таблицы *index_of_element*, задающих взаимное отображение:

```
record_element (e: G)
    - Добавить e в множество элементов, если там его еще нет.
  require
    not_sorted: not done
  do
    if not has_element (e) then
          count:= count + 1
          index_of_element. extend (count, e)
          element_of_index. force (e, count)
    -extend и force расширяют структуры при необходимости; это означает,
    -что нам не требуется знать, сколь много элементов может появиться.
    end
  ensure
          inserted: has_element (e)
          one_more: not (old has_element (e)) implies (count = old count + 1)
  end
```

Начальный тест должен убеждать, что процедура игнорирует повторную попытку вставки данного элемента. Эта политика позволяет *record_constraint*(*e*, *f*) стартовать, вызывая *record_element* как на *e*, так и на *f*, просто для того, чтобы убедиться, что элементы вставлены надлежащим образом. В упражнении вас попросят найти способ, позволяющий избежать дублирования работы между *has_element* и *extend*.

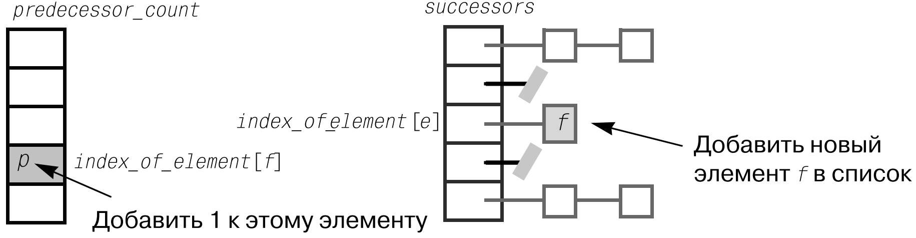


Рис. 15.18. Добавление ограничения

При подходящей реализации *extend* и *force* код процедуры *record_element* выполняется за O(*1*), что для всех элементов дает время O(*n*). Это согласуется с нашими требованиями.

Оставшийся механизм инициализации дается процедурой для ввода ограничений. Вызов *record_constraint*(*e*, *f*) должен увеличивать на 1 число предшественников *f* в массиве *predecessor_count* и добавлять *f* в список последователей *e*. Этот список является одним из элементов массива *successors:*

Вот текст процедуры:

```
record_constraint (e, f: G)
      – Добавить ограничение [e, f]
    require
      not_sorted: not done
      exist: e /= Void and f /= Void
    local
      x, y: INTEGER
    do
      – Убедиться, что e и f вставлены (нет эффекта, если они уже там присутствуют):
      record_element (e); record_element (f )
      x:= index_of_element [e]
      y:= index_of_element [f ]
      predecessor_count [ y]:= predecessor_count [ y] + 1
      add_successor (x, y)
ensure
      both_there: has_element (e) and has_element (f)
end
```

Дополнительная процедура, которую можно не экспортировать:

```
add_successor (x, y: INTEGER)
      – Запись y как последователя x.
    require
      1 <= x; x <= count
      1 <= y; y <= count
    local
      x_successors: LINKED_LIST [INTEGER]
    do
      x _successors:= successors [x]
– Список последователей для x может быть еще не создан:
      if x_successors = Void then
                        create x_successors. make
                        successors [x]:= x_successors
      end
      x_successors. extend (y)
    end
```

Как уже отмечалось, работа *record_constraint* начинается с двух вызовов *record_element* для пары аргументов, задаваемых в ограничении. Из-за способа проектирования *record_element* эффекта от вызова не будет, если элементы уже присутствовали. Эта политика делает воз-

можным для клиентского приложения начинать непосредственно со списка ограничений, никогда явно не обращаясь к записи элементов.

> Мы не можем, однако, полагать, что так будет всегда, и удалить *record_element* из интерфейса, доступного клиенту. Для некоторых вариантов задачи, о чем ранее говорилось, в исходном множестве элементов могут встречаться такие элементы, которые не входят ни в одно из ограничений. Понятно, что после топологической сортировки множества они являются полноценными участниками вывода. Для таких вариантов ввод должен включать помимо списка ограничений и список элементов, по крайней мере, тех, что не входят в ограничения.

Говоря о дублировании, заметим, что процедура *record_constraint* не пытается определить, встречалось ли уже вводимое ограничение. Можете убедиться, что алгоритм сортировки корректно будет работать и в случае, когда ограничение повторяется. Применение другой политики, запрещающей дублирование, возлагается на клиента, ответственного за ввод данных.

Вернемся к эффективности. Код каждой из двух дополнительных процедур выполняется за время O(1): доступ к элементу массива, запись в конец списка во втором случае (при хорошей организации списка с курсором в конце операция выполняется за константное время). Так что время работы *record_constraint* также задается O(1), а поскольку процедура должна отработать для каждого ограничения, в целом получаем $O(m)$. Таким образом, достигнута наша цель: получить алгоритм, выполняющийся за время $O(m + n)$ как на этапе инициализации, так и при выполнении основной задачи – топологической сортировки.

Собираем все вместе

Мы уже познакомились со всеми элементами нашей программы, необходимыми для реализации топологической сортировки. Класс, построенный в полном соответствии с данным обсуждением, доступен в EiffelBase и используется в Traffic, но, полагаю, для проверки вашего понимания концепций следует самостоятельно написать его реализацию, собирая все вместе.

Время программирования!

Реализация топологической сортировки
Напишите класс *TOPOLOGICAL_SORTER*, обеспечивающий универсальную, практичную топологическую сортировку.

Убедитесь, что решение отвечает принципам инженерии программ, не только обеспечивает эффективный алгоритм, но и включает процедуры инициализации (*record_element, record_constraint*). Для тестирования решения используйте файл, доступный на сайте, который связан с курсом. Этот файл содержит несколько сотен ограничений и все возможные варианты топологической сортировки.

15.5. Уроки

Из алгоритма топологической сортировки можно вывести важные следствия, общезначимые для проектирования алгоритмов и для программной инженерии в целом.

Интерпретация в сравнении с компиляцией

Мы уже рассматривали два стиля выполнения программ, написанных на некотором языке программирования S.

- Интерпретация. Написать программу, называемую **Интерпретатором**, которая может выполнять произвольную *S*-программу для произвольных входных данных.
- Компиляция. Написать программу, называемую **Компилятором**, которая может преобразовать произвольную *S*-программу в программу с эквивалентной семантикой, записанную на целевом языке *T*. Если *T* – это машинный язык для применяемой платформы, то результат компиляции может быть непосредственно выполнен. В противном случае *T*-программа может быть интерпретирована или подвергнута дальнейшей компиляции.

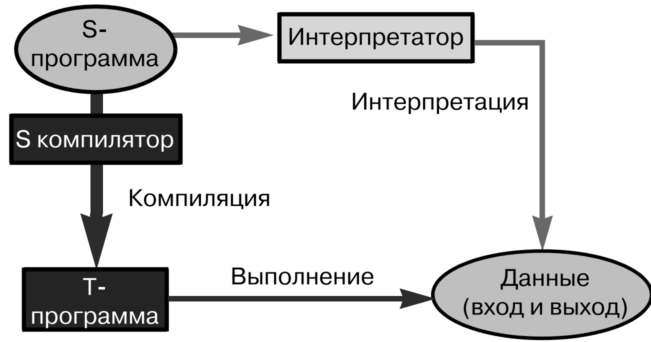


Рис. 15.19. Интерпретация и компиляция программ

Как отмечалось, практическая реализация языков часто комбинирует оба подхода.

Эти концепции обобщаются на многие проблемные области, отличающиеся от выполнения программ, которые написаны на языках высокого уровня. Для выполнения некоторой обработки на некотором входе мы можем использовать структуры данных, которые непосредственно отображают вход, или можем разбить процесс обработки на два этапа.

- Компиляция: преобразовать данные в форму, более подходящую для целей алгоритма.
- Интерпретация: применять требуемые операции к исходной структуре данных.

Этот прием (который, как и при обработке языков, может и в общем случае включать несколько итераций процесса) в точности был применен для топологической сортировки. Вначале мы рассмотрели интерпретационное решение, которое использует кажущиеся естественными структуры данных, непосредственно следуемые из постановки задачи. Но оказалось, что эти структуры приводят к «плохому» по производительности решению. «Компиляция» их в представление, подогнанное под наши цели, привело к решению с прекрасной производительностью.

В таком двухэтапном решении шаг «компиляции», инициализирующий структуры данных, может быть столь же сложным для проектирования, как и фактическая обработка, основанная на этих структурах, – он может требовать столь же много времени, а иногда и больше, чем время самой обработки. Этот подход хорош, если общая производительность соответствует вашим целям, но, конечно же, нельзя перепрыгивать к заключению о производительности алгоритма, не учитывая время инициализации, а не только самой обработки.

Исходные данные
Инициализация
«Компилятор»
«Компилированная» форма
Основной алгоритм
«Интерпретатор»
Вывод

Рис. 15.20. «Интерпретация» и «компиляция» данных

Подход может быть обобщен в виде эвристики – универсальной стратегии, подобной «образцу проектирования», но в существенно более абстрактной форме.

Почувствуй эвристику

Вначале компилируй данные!

Хорошие алгоритмы часто создаются с использованием двухэтапной стратегии, где:

- на первом этапе преобразуется вход из его исходной формы во внутренние структуры, тщательно спроектированные и подогнанные под цели задачи;
- на втором шаге обрабатываются преобразованные данные для получения конечного результата.

Мы увидим еще одно применение этой идеи при обсуждении архитектуры процесса проектирования, управляемого событиями.

Компромисс «время-память»

Выбора тяжко бремя – память иль время!

Тесно связанным с эвристикой «Вначале компилируй данные» является наблюдение, что идеальная структура данных, та, которая наилучшим способом помогает реализации второго шага, часто не является наиболее экономичным по памяти представлением информации. В топологической сортировке информация об ограничениях может сохраняться в трех различных частях. Ограничение [*x*, *y*] ведет к тому, что *y* появится также в списке последователей *x* в массиве *successors*; некоторая информация хранится в массиве *predecessor_count*[*y*]. Если для заданного *y* ограничения отсутствуют, то это приводит к добавлению элемента в массив *candidates*. При таком представлении нам приходится жертвовать памятью для достижения требуемой по времени производительности. Компромиссы такого вида в ту или иную сторону – ключевые моменты проектирования эффективных алгоритмов.

Алгоритмы в сравнении с программами и компонентами

Вполне возможно было бы привести описание алгоритма топологической сортировки, которое игнорирует многие из аспектов, рассмотренных в этой главе, концентрируясь только на

процессе сортировки. Для практического решения необходимо принимать в расчет практические потребности приложений. ОО-подход позволяет нам действовать в соответствии с этими целями. Вместо написания *процедуры* топологической сортировки мы проектируем *класс TOPOLOGICAL_SORTER*. Экземпляр этого класса характеризует задачу топологической сортировки, предоставляя не только алгоритм ее решения (процедуру *process*), но и весь инструментарий, необходимый клиенту для того, чтобы он мог:

- поставить задачу, записав элементы и ограничения удобным для себя способом;
- применить *process* для выполнения топологической сортировки заданных элементов, удовлетворяющих заданным ограничениям;
- запросить данные о результирующем состоянии, выяснить, существуют ли циклы, и если да, то получить информацию об элементах, входящих в цикл.

Разница между этим подходом и чистым алгоритмом является частью разницы между **программной инженерией** и чистым программированием. В инженерии программ недостаточно спроектировать умный алгоритм и связанные с ним структуры данных. Целью становится подготовка решения, которое может быть интегрировано в успешные программные системы.

Наши решения должны допускать **повторное использование**, так что они не просто представляют «образцы проектирования», которые программист может интегрировать в свою систему, купив и прочитав книги (особенно выдающиеся книги, такие как эта, например). Решения должны быть реализованы как компоненты, доступные всегда и для всех, непосредственно доступные, а не стоящие на полках.

Вы также обратили внимание на то, как в этом процессе контракты позволяют нам точно знать, что нам следует делать, – что мы ожидаем, что гарантируется и что мы поддерживаем.

15.6. Ключевые концепции этой главы

- Топологическая сортировка является перечислением множества элементов, которое совместимо с множеством ограничений порядка, заданного на этих элементах.
- Задача имеет простое математическое описание: задано (строгое) отношение порядка, требуется найти полное отношение, чьим подмножеством является исходное отношение.
- На практике отношение обычно является не отношением порядка, а в лучшем случае ациклическим отношением. Взятие транзитивного замыкания дает отношение порядка.
- Реалистичное, хорошее инженерное решение должно принимать, возможно, ошибочный вход, содержащий циклы. Оно должно выполнять топологическую сортировку ацикличной части исходного отношения и дополнительно выдать информацию об элементах, входящих в цикл.
- Такое решение должно предоставлять не только алгоритм топологической сортировки, но и механизмы построения экземпляра задачи с вводом элементов и ограничений.
- Для n элементов и m ограничений топологическую сортировку можно выполнить за $O(m + n)$ время, используя память того же порядка.
- Ключом эффективности алгоритма является построение структур данных, специально приспособленных для решения задачи сортировки:
  - массив, который для каждого элемента исходного множества содержит список его последователей;
  - массив, который для каждого элемента исходного множества содержит число его предшественников;

- распределитель (стек, список или очередь с приоритетами), который содержит множество элементов, не имеющих предшественников.

- На примере задачи сортировки показано, что хорошее алгоритмическое решение часто получается путем «компиляции» исходных данных задачи в специально спроектированные структуры данных, которые затем могут быть эффективно «интерпретируемы».

Новый словарь

Acyclic	Ациклический	**Antisymmetric**	Антисимметричность
Asymmetric	Асимметричность	**Binary relation**	Бинарное отношение
Cycle	Цикл	**Irreflexive**	Иррефлексивность
Order relation	Отношение порядка	**Partial order**	Частичный порядок
Relation	Отношение	**Strict order**	Строгий порядок
Topological sort	Топологическая сортировка	**Total order**	Тотальный или полный порядок
Transitive closure	Транзитивное замыкание		

15.7. Приложение. Терминологические замечания об отношениях порядка

Для обсуждения топологической сортировки удобно — как показано в этой главе — использовать отношение строгого порядка: «строго меньше, чем». Классическим примером такого отношения является отношение «<», заданное на числах. В некоторых задачах может быть более полезным иметь дело с нестрогим отношением, как отношение «<=» для чисел. Пара элементов связана нестрогим отношением $x <= y$, если и только если либо $x < y$, либо $x = y$. Общепринятое соглашение для «отношения порядка» предполагает нестрогую версию. В этой главе, так как мы использовали только строгое отношение порядка, слово «строгое» обычно опускалось, так что «порядок» означал «строгий порядок».

Для отношения строгого порядка (иррефлексивного и транзитивного) в литературе иногда используется термин «квазипорядок». Конечно, каждый может дать любое имя понятию, задав его точное определение, но это конкретное имя неудачно, так как ничего «квази» в таком порядке нет. Если и есть что-нибудь характерное для элементов в этом порядке, то это то, что они «более» упорядочены, чем для случая нестрогого порядка, поскольку имеет место иррефлексивность, запрещающая элементу находиться в отношении с самим собой. Чтобы совсем запутать дело, некоторые авторы используют термин «квазипорядок» для отношений, которые являются рефлексивными и транзитивными. Так что лучше этот термин «квази» не использовать и вместо этого квалифицировать отношение порядка как «строгое», когда это необходимо.

Еще одна проблема: является ли отношение тотальным или нет. Тотальность означает, что для любых пар различных элементов x и y имеет место одно из двух — либо $x < y$, либо $y < x$. Отношение порядка, удовлетворяющее этому свойству, называется тотальным порядком. Частичным порядком должно было бы называться отношение, которое не удовлетворяет этому свойству, такое, что для него найдется по крайней мере одна пара различных элементов, для которой ни $[x, y]$, ни $[y, x]$ не принадлежат отношению. Но не так определяется частичный порядок в большинстве литературных источников — там частичный порядок определяется как такой порядок, о котором неизвестно, является ли он тотальным. Другими словами, это отношение *возможно частичного* порядка. Это приводит к путанице, так как теперь

тотальный порядок является также частичным порядком! Поэтому лучше писать так, как принято в этой главе:

тотальный порядок – в тех случаях, когда известно, что он тотальный;

частичный порядок – для отношения порядка, о котором известно, что порядок не является тотальным.

Если же мы не знаем, каков порядок, или хотим включить оба случая – лучше применять термин «*порядок*». В случае неопределенности –использовать термин «*возможно частичный порядок*».

15-У. Упражнения

15-У.1. Словарь

Дайте точные определения символам словаря.

15-У.2. Иррефлексивность и асимметричность

Отношение порядка было определено как транзитивное и иррефлексивное. Было доказано, что следствием является асимметричность отношения. Докажите, что если отношение определяется как транзитивное и асимметричное, то следствием является иррефлексивность.

15-У.3. Тотальный порядок и перечисление

Докажите, что если отношение r является отношением строгого тотального порядка на конечном множестве, то существует единственное перечисление элементов, такое, что для любых элементов x и y элемент x появится перед y в перечислении, если и только если пара $[x, y]$ принадлежит r.

15-У.4. Строгое отношение в сравнении с нестрогим отношением порядка

Обсуждение в этой главе строилось на строгом отношении порядка (частичном или полном), таком, как отношение «<» на числах – «меньше чем». Также возможно использовать нестрогое отношение порядка, такое, как «⩽» – «меньше или равно». Определение строгого частичного порядка, которое мы будем называть отношением «<», хотя оно и не должно быть обычным отношением на числах, должно удовлетворять свойствам:

O1 **иррефлексивности**: $x < x$ не выполняется для любого x.
O2 **транзитивности**: из того, что $x < y$ и $y < z$, следует $x < z$.

Для этого отношения также выполняется свойство:

O3 **асимметричности**: не могут одновременно иметь место $x < y$ и $y < x$.

Свойство O3 является следствием двух предыдущих свойств.
Для *частичного нестрогого* отношения порядка «⩽» имеют место три независимых свойства:

N1 **рефлексивности**: $x \leqslant x$ для любого x.
N2 **транзитивности**: из того, что $x \leqslant y$ и $y \leqslant z$, следует $x \leqslant z$.
N3 **антисимметричности**: из того, что $x \leqslant y$ и $y \leqslant x$, следует $x = y$.

Для любого частичного строгого отношения порядка «<» существует связанное с ним нестрогое отношение «⩽», определенное как

```
x ⩽ y если и только если: x < y или x = y                                    [4]
```

Справедливо и обратное утверждение: для любого частичного строгого отношения порядка «⩽» существует связанное с ним строгое отношение «<», определенное как

```
x < y если и только если: x ⩽ y и x ≠ y                                      [5]
```

В этом упражнении требуется исследовать связь между этими ассоциированными отношениями – строгим «<» и нестрогим «⩽».

1. Докажите, что если «<» является частичным строгим отношением порядка, то «⩽», определенное в **[4]**, является частичным нестрогим отношением порядка.
2. Докажите, что если «⩽» является частичным нестрогим отношением порядка, то «<», определенное в **[5]**, является частичным строгим отношением порядка.
3. В строгом случае определение требует выполнения только двух свойств: иррефлексивности и транзитивности. Третье свойство асимметричности является следствием двух остальных. В нестрогом случае требуется выполнение всех трех свойств. Докажите, что антисимметричность не является следствием двух остальных. Другими словами, приведите пример нестрогого отношения, для которого первые два свойства выполняются, а антисимметричность не имеет места.
4. Докажите, что замена «рефлексивности» на «иррефлексивность» в определении нестрогого порядка приводит к определению строгого порядка.
5. Приводит ли замена «иррефлексивности» на «рефлексивность» в определении строгого порядка к определению нестрогого порядка?

15-У.5. Ацикличность и отношение тотального порядка

Докажите теорему о топологической сортировке, используя вторую теорему об ацикличности и отношениях порядка.

15-У.6. Интересное постусловие

Объясните постусловие функции *has_element*.

15-У.7. Оптимизация использования хэш-таблицы

В процедуре *record_element* проверяется, присутствует ли уже вставляемый элемент в таблице *index_of_element*, и запись осуществляется только в случае, когда такого элемента нет. В реализации используются две операции поиска – одна при вызове *has_element*, другая при вызове *extend*. Проанализируйте контрактную форму класса *HASH_TABLE*, найдите нужные компоненты, перепишите процедуры так, чтобы устранить эту небольшую неэффективность.

15-У.8. Программирование топологической сортировки

Постройте класс *TOPOLOGICAL_SORTER* в соответствии с обсуждением этой главы.

15-У.9. Параметризация топологической сортировки

(Это упражнение предполагает, что выполнено предыдущее)

Расширьте реализацию топологической сортировки, позволяя клиентам задавать стратегию выбора элемента из множества соревнующихся «кандидатов».

Часть IV

Объектно-ориентированные приемы программирования

В этой четвертой части книги переместимся на передний край современной технологии программирования и посмотрим, какие преимущества можно получить, если использовать всю мощь ОО-идей.
Первая из трех глав описывает многие механизмы наследования, включая полиморфизм, динамическое связывание, множественное наследование, понятие ограниченной универсальности. Вторая глава исследует новые средства, которые добавляют существенную выразительную силу объектному каркасу. В Eiffel они называются агентами (терминология не устоялась, в других языках их называют делегатами, замыканиями). Рассмотрение агентов сопровождается знакомством с основами лямбда-исчисления. Последняя глава этой части представляет проектирование, управляемое событиями, – гибкую архитектуру программных проектов, дополняющую наши прежние структуры управления.

16

Наследование

Мир (готовы ли вы уже в самом начале главы погрузиться в водовороты жизни?) полон беспорядка. Возможно, Природа ненавидит беспорядок, а может быть – и нет, я в этом не уверен. Ваша точка зрения во многом зависит от того, кого вы читали – Платона, Аристотеля, Канта. Наука определенно борется с беспорядком. Здравые рассуждения о мире требуют порядка.

Наука создает порядок: идеализированную, формализованную версию реальности. Оставим друзьям-философам споры о том, присутствовал ли порядок с сотворения мира (тогда все, что должна сделать наука – это обнаружить этот порядок), или мир изначально неупорядочен и наука лишь пытается навести искусственный порядок в естественной хаотичности мира.

Что касается нас, то мы займемся пониманием наследования – наукой, инженерией, поиском систематических, хорошо структурированных описаний. Одним из важнейших инструментов, помогающих в этом поиске, является иерархическая классификация, известная также как таксономия.

Чтобы стать науками, ботанике и зоологии нужен был Линней, который в 18-м столетии создал эффективную классификацию живых существ. Биологическая таксономия говорит нам, что дельфины принадлежат *виду*, называемому по латыни *Delphinius delphis*, они включены в *род Delphinius,* который сам является частью – пропуская некоторые уровни класси-

фикации – *отряда* китовых из *класса* млекопитающих, принадлежащего, вне всякого сомнения, *царству* животных[1]. Объекты в этом случае естественные, а классификация искусственная.

> Действительно, Линнеевская классификация – одна из многих возможных. Аристотель, который задолго до Линнея ввел свою классификацию, не объединял дельфинов с млекопитающими, живущими на суше, хотя он и осознавал, что они не относятся к рыбам, которых он классифицировал ранее.

Объекты в математике – числа, функции, последовательности – являются искусственными, плодами человеческого воображения. Эварист Галуа в начале 19-го века, а затем Георг Кантор и другие для группировки таких математических объектов предложили абстрактные математические структуры, объединяя объекты в категории, создающие иерархическую структуру. Объекты, для которых определена единственная ассоциативная операция и выделен объект с особыми свойствами, называемый единицей, образуют *моноид*. Примером моноида могут служить строки, где в качестве операции рассматривается конкатенация строк, а единицей является пустая строка. Другим примером являются целые неотрицательные числа с операцией сложения и нулем в качестве единицы.

Если в моноид добавить новое свойство, получим *группу*. Группа является моноидом, в котором у каждого элемента существует обратный элемент. Примером группы могут служить множество целых чисел с операцией сложения, с нулем в качестве единицы группы, и для каждого элемента x в группе существует обратный элемент $-x$. Добавив в группу еще одну операцию, можем получить *кольцо*. Примером кольца могут служить целые числа с операциями сложения и умножения. Добавление новых свойств позволяет получить *поле* (пример – поле действительных чисел).

Подобно математикам, мы, программисты, имеем дело с абстрактными объектами – творениями нашего воображения – и не можем обвинять никого, кроме себя, при обнаружении беспорядка в мире наших объектов. И нам, как и всем, нужно преобразовать беспорядок, создавая подобие порядка. Возможно, нам это нужно *больше*, чем кому-либо, поскольку мы являемся чемпионами в создании энтропии, ибо наши программы способны создать самую немыслимую путаницу, которая только возможна. Говорят, что «*человеку свойственно ошибаться, но путаницу несомненно создает компьютер*»; добавим к этому: «*или компьютерный программист*».

Чтобы справиться с беспорядком, подобно всем наукам, мы можем использовать таксономию. Мы можем объединить наши объекты в категории и рассматривать иерархические отношения между этими категориями. Как дельфин, будучи млекопитающим, *является* позвоночным животным, как поле в математике *является* кольцом, так и такси, моделируемое в нашей программе, *является* транспортным средством и как таковое *является* движущимся городским объектом. Пешеход также *является* движущимся городским объектом, но *не является* транспортным средством. Наследование позволит нам делать выводы о таких отношениях, как «*является*» («is–a») и использовать таксономии для структурирования нашего ПО.

[1] Основная классификация Линнея обычно содержит 7 уровней: царство -> тип -> класс -> отряд -> семейство -> род -> вид. Подробная классификация может содержать более 30 уровней, вводя имена промежуточных уровней и создавая уровни с использованием приставок – подтип, надкласс, суперкласс, подотряд и так далее. Для именования объектов используются два последних уровня род и вид. Мы, люди, согласно этой классификации относимся к Homo sapiens (род – человек, вид – разумный). Как и дельфины, мы входим в класс млекопитающих из царства животных.

Мы уже встречались с наследованием и связанным с ним ключевым словом **inherit** – фактически уже в самом первом примере – как со способом получения в нашем классе преимуществ от работы, выполненной в другом классе. Но это только один из аспектов наследования, который применим к классу, рассматриваемому как модуль – собрание под одной обложкой полезных компонентов. Наследование становится куда более интересным – за счет новых приемов, таких как полиморфизм и динамическое связывание – для приложений, где используется другая роль классов: роль *типов* данных. В этой роли класс описывает коллекцию объектов периода выполнения, таких как линии метро или такси.

16.1. Такси и транспортные средства

Да, такси. В предыдущих главах мы отдали дань нашим простонародным привычкам и путешествовали в метро, где кого только не встретишь. Сейчас же поднимемся на следующую ступеньку и будем распоряжаться собственным транспортным средством.

Наследуемые компоненты

Класс TAXI из TRAFFIC обеспечивает – вы может это проверить – такой компонент, как

```
take (from_location, to_location: LOCATION)
                    -- Доставить пассажира из from_location в to_location
```

Другим компонентом класса является *office*, представляющий диспетчерский офис службы такси.

При чтении текста класса вы увидите только небольшое число других компонентов. В то же время у такси свойств значительно больше. Например:

- такси имеет пассажиров (в противном случае комментарий для компонента take не имел бы смысла: кого следует доставить из одной точки в другую?). Класс должен иметь команду для посадки пассажиров и запрос, позволяющий выяснить текущее число пассажиров;
- в любой момент такси имеет текущую позицию.

Где же находятся соответствующие свойства? Ответ можно найти, взглянув в начало объявления класса:

```
note
…
class
    TAXI
inherit
    VEHICLE
feature
    … Оставшаяся часть класса …
```

Класс *TAXI* наследует от *VEHICLE*; и в самом деле, если посмотреть на класс *VEHICLE,* то можно найти команды *load*, для посадки пассажиров в транспортное средство, так же как и *unload* – для высадки, и запрос *count*, дающий текущее число пассажиров. Теперь, обратившись к началу объявления класса *VEHICLE,* вы увидите:

```
note
…
deferred class
      VEHICLE
inherit
      MOVING
feature
… Оставшаяся часть класса …
```

Класс *VEHICLE* наследует от класса *MOVING*, который описывает движущиеся объекты и имеет запрос *position*.

Классы *VEHICLE* и *MOVING* объявляются не просто как **class**, а как **deferred class**. Мы вскоре познакомимся детально с концепцией отложенного класса, указывающего на то, что не все его компоненты полностью заданы; реализация некоторых из них оставлена его потомкам – классам, наследующим от него.

Глядя, как эти три класса описывают типы объектов периода выполнения – такси, транспортные средства, движущиеся объекты, – мы понимаем, о чем говорит нам наследование. Оно устанавливает, что в системе Traffic любое такси может рассматриваться как транспортное средство, которое, в свою очередь, является движущимся объектом. В частности, все свойства класса *MOVING* применимы к целям типа *VEHICLE* и *TAXI*, и все свойства класса *VEHICLE* применимы к целям типа *TAXI*.

Термины наследования

Как обычно, помогает точная терминология.

Определения: наследник, родитель, (правильный) потомок и предок

> Если *B* наследует от *A* (В Eiffel *A* перечислено в предложении **inherit** класса *B*), то *B* наследник *A*, а *A* – родитель *B*.
> Потомками класса является сам класс и (рекурсивно) потомки его наследников. Сам класс не включается в число правильных потомков. Предок и правильный предок являются обращенными понятиями по отношению к потомкам.

Определение потомков рекурсивно, но это теперь не должно вас смущать. Неформальный способ этого определения говорит, что потомком класса является сам класс, его потомок, потомок его потомка и так далее.

В литературе иногда встречается термин «подкласс», означающий иногда наследника, иногда правильного потомка. Аналогично встречается и термин «суперкласс».

На рисунке ниже, иллюстрирующем наш пример, все классы являются потомками класса *MOVING*. Все классы – его правильные потомки, за исключением самого класса. Правильными предками класса *TAXI* являются классы *VEHICLE* и *MOVING*.

Рядом с каждым классом на рисунке показан один или два компонента, вводимые этим классом. Обратите внимание на соглашение, принятое при представлении наследования на диаграммах: наследование изображается в виде одиночной стрелки, Напоминаю, что другое отношение – «клиентское» – изображается двойной стрелкой.

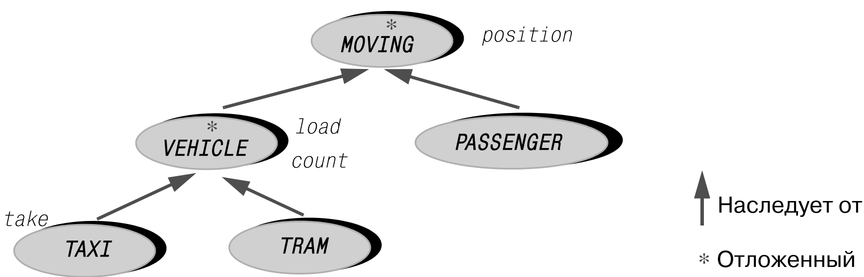


Рис. 16.1. Иерархия наследования

Направление стрелки, идущее от наследника к родителю, является напоминанием проектировщику, что ему все известно о родительском классе и предках, но ничего не известно о правильных потомках класса.

От вас не требуется вручную рисовать диаграмму, отображающую отношения между классами. Если классы скомпилированы, то инструментарий Diagram Tool EiffelStudio создаст диаграмму. Достаточно просто щелкнуть вкладку Diagram, и появится требуемая диаграмма:

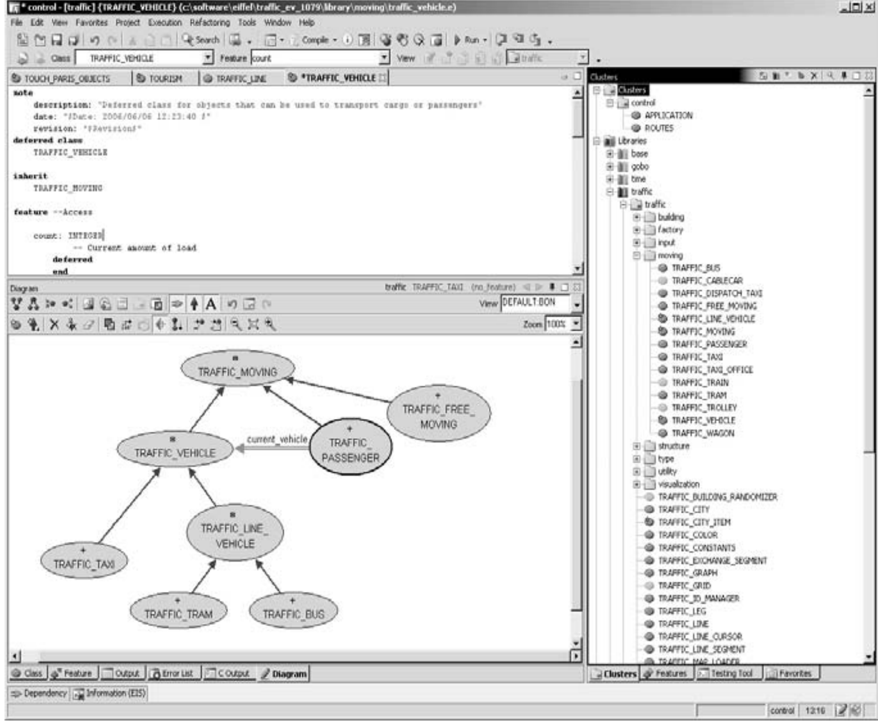


Рис. 16.2.

Если класс не существует, но вы занимаетесь его проектированием, то также можно пользоваться этим инструментарием. Можно графически описать классы и задать связи (наследования и клиентские) между ними. В этом режиме будут автоматически сгенерированы соответствующие тексты классов.

Компоненты, приходящие от высших авторитетов

Мы можем теперь оценить новинку, вводимую наследованием. Понятие «компоненты класса» больше не означает только компоненты, заданные в классе, но и компоненты, наследуемые от родителя. Так, объявив объекты m: *MOVING, v: VEHICLE, t: TAXI*, мы можем помимо прочего писать:

```
v.load (…)
t.take (…)
v.count                                              -- Выражение
```

Возможны и другие вызовы, такие как:

```
t.count
t.load
t.position

v.position
```

Здесь используются компоненты, наследуемые от родителя или, более точно, от правильных предков. Полезно различать «наследуемые» и «непосредственные» компоненты.

Определения: компоненты класса, непосредственные, наследуемые, вводимые

Компонент класса – это одно из двух:

- наследуемый компонент, если это компонент одного из родителей класса;
- непосредственный компонент, если он объявлен в классе и не наследуется. В этом случае говорят, что класс вводит компонент.

Заметьте, что определение включает рекурсию.

Отныне следует понимать, что класс *A* больше, чем то, что видимо в тексте класса. Компоненты класса означают не только компоненты, объявленные в тексте класса, но и те, что приходят от родителей, если таковые существуют.

Плоский облик

Как тогда получить полную картину? Плоским обликом класса называется искусственно сконструированная версия, которая включает все компоненты, непосредственные и наследуемые. Это не то, что вы пишете, создавая класс, но лишь облик, подобный контрактному облику, который, как мы видели, дает нам свободную от реализации версию класса. EiffelStudio создает этот взгляд на класс, этот облик, для чего достаточно выбрать класс и щелкнуть по кнопке «Flat view»:

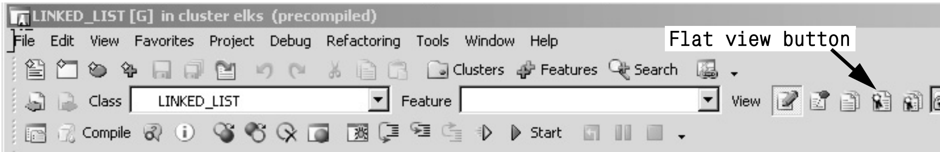


Рис. 16.3.

Результат похож на обычный текст класса (с некоторой новой нотацией, которая позже прояснится). Отметьте появление нового вида комментариев, здесь для компонента *LINKED_LIST*:

```
LINKED_LIST
                end
                go_to (p)
            end
        end

    index_of (v: like item; i: INTEGER_32): INTEGER_32
            -- Index of `i'-th occurrence of item identical to `v'.
            -- (Reference or object equality,
            -- based on object_comparison.)
            -- 0 if none.
            -- (from CHAIN)
        require -- from LINEAR
            positive_occurrences: i > 0
```

Рис. 16.4. Плоский облик

Выделенных комментариев нет в исходном тексте, но они добавлены EiffelStudio, когда он создает плоский облик. Они указывают, что компонент наследуется, приходя от правильного предка *CHAIN*, и что его предусловие было определено в другом предке – *LINEAR*. Мы вскоре увидим, как контракты преобразуются в связи с наследованием.

Контрактный облик класса выводится (как объяснялось ранее, удаляется закрытая информация) не из исходного текста класса, а из его плоского облика. Для клиента класса обычно все равно, наследованы ли экспортируемые классом компоненты или введены самим классом. Интерфейсный облик класса позволяет ограничиться только непосредственными компонентами.

16.2. Полиморфизм

Аккумулирование компонентов не является единственной целью наследования. Мы уже демонстрировали эту возможность, когда писали класс PREVIEW как наследника класса TOURISM. Мы хотим получить преимущества от наследования не просто за счет модульности, а за счет того, что оно задает отношения между типами, – дельфины *являются* млекопитающими, кольца *являются* группами.

Переходя к программистскому примеру: из того, что «любое такси *является* транспортным средством», следует возможность присваивания между переменными и выражениями этих двух типов. Пусть объявлены переменные:

```
my_vehicle: VEHICLE
cab_at_corner: TAXI
```

Структура наследования, рассмотренная выше, делает допустимым присваивание:

```
my_vehicle := cab_at_corner
```

Оно отличается от присваивания, которое нам встречалось до сих пор. Ранее выражение источника присваивания и переменная, задающая цель присваивания, были одного типа, но теперь они различны. Специфика в том, что тип источника является потомком типа цели.

В эффекте присваивания нет ничего нового. Все, что мы изучили о присваивании, сохраняется, если не считать изменений в типе объектов. Предполагая, что обе переменные первоначально присоединены к объектам, и картинка, отображающая эффект «до-после» присваивания, полностью сохраняется:

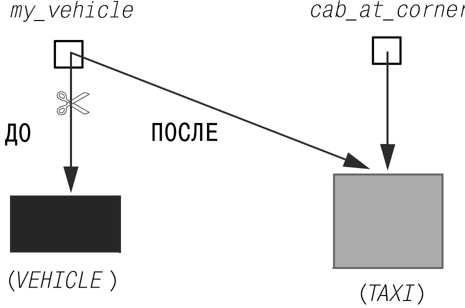


Рис. 16.5. Полиморфное присваивание

Это обычное ссылочное присваивание. Сами объекты – здесь объекты типов *TAXI* и *VEHICLE* – не меняются. Новинка в том, что после присваивания переменная типа *VEHICLE* теперь может быть присоединена к объекту типа *TAXI* – к объекту одного из своих потомков.

Определения

Нам нужна подходящая для этой ситуации терминология.

Определения: полиморфизм

Присоединение (присваивание или передача аргумента) является полиморфным, если целевая переменная и выражение источника имеют различные типы.

Сущность или *выражение* является полиморфным, если – как результат полиморфного присоединения – она может во время выполнения быть присоединена к объектам различных типов.

Контейнерная структура данных является полиморфной, если она может содержать ссылки на объекты различных типов.

Напоминаю, что «сущности» включают переменные (атрибуты, локальные переменные), а также формальные аргументы методов и Result.

«Полиморфизм» лежит в основе этих возможностей. Его часто путают с *динамическим связыванием*, изучаемым ниже. Динамическое связывание необходимо для реализации полиморфизма, но это разные концепции.

Как отмечалось в определении, полиморфизм существует не только при присваивании, но и при передаче аргумента в момент вызова метода. Пусть некоторый произвольный класс, например, *DAILY_SCHEDULE*, имеет метод:

```
register_trip (v: VEHICLE)
```

Тогда вызов этого метода является вполне корректным:

```
register_trip (cab_at_corner)
```

Здесь тип фактического аргумента является потомком типа формального аргумента. Наиболее интересно здесь то, что, когда пишется метод, такой как *register_trip*, то используется не полное, а частичное знание. Автор знает, что во время выполнения значение аргумента будет присоединено к объекту, представляющему некоторый вид транспортного средства – *VEHICLE,* но этот объект может быть *TAXI*, или *TRAM*, или любым другим транспортным средством, и автор метода не знает, каким именно, и ответ может изменяться от одного выполнения к другому.

Некоторые из соответствующих классов могут быть написаны уже после выпуска заключительного релиза метода *register_trip*. Подумайте: что, если бы вам пришлось писать такой метод, понимая, что аргументы метода могут быть во время выполнения связаны с объектами, классы которых еще и не спроектированы! Динамическое связывание позволяет справиться с этим вызовом.

Полиморфизм – это не трансформация

Несмотря на свое имя – от греческого словосочетания «множественность форм» – полиморфизм не является причиной изменения во время выполнения объектом своей «формы» (изменением типа). Полиморфное присоединение применимо только к ссылочным типам с эффектом, показанным на последнем рисунке, – изменяются ссылки, но *сами объекты не меняют тип*.

Иногда возникает необходимость трансформации объекта, но это никак не связано с полиморфизмом. Простейшим примером такой ситуации является присваивание целого целевой переменной типа *REAL*, чье внутреннее представление отличается от представления источника. Подходящим механизмом в таком случае является **трансформация**, но не полиморфное присоединение.

Общий механизм трансформации, применимый как к ссылочным, так и к развернутым типам, механизм, который стоит за возможностью присваивания целых вещественным переменным, поддерживается в Eiffel. Язык позволяет также определять собственную трансформацию между создаваемыми типами данных. Если проектируется класс *DATE* с атрибутами *day*, *month*, *year*, то в класс можно включить метод, осуществляющий преобразование из *DATE* в *STRING*, что позволит преобразовать дату и представить ее в виде строки текста (например, в виде «13.06.2010» или в любом другом формате, выбранном в методе преобразования).

Мы не будем более останавливаться на механизмах преобразования. Если необходимо его использовать, то можно проанализировать для начала класс *REAL_32* в EiffelBase (смотри предложение **convert**), этого будет достаточно для понимания основных идей трансформации.

Что же касается нашего обсуждения, то следует понимать, что трансформация и полиморфизм являются взаимоисключающими механизмами: если применяется один из них, то второй не применяется. Так что, когда вы видите присваивание *a*:=*b* или когда речь идет о передаче аргумента при вызове метода, то никогда не возникает неопределенность, из контекста всегда ясно, с каким случаем мы имеем дело.

Полиморфные структуры данных

Особый интерес представляет последний случай в определении полиморфизма – полиморфная структура данных, называемая также полиморфным контейнером. Рассмотрим типичный контейнер – список, предназначенный для хранения транспортных средств:

```
fleet: LIST [VEHICLE]
```

Рассмотрим вызов, такой как *fleet.extend*(...), добавляющий элемент в список. Какой вид аргументов можно использовать в вызове, заменяя «...»? Если посмотреть на объявление *extend* в *LIST*[*G*], то можно видеть:

```
extend (v: G)
            -- Добавить новое вхождение v в конец.
```

В случае с *fleet* фактическим родовым параметром, соответствующим *G*, является тип *VEHICLE*, так что вызов ожидает аргумента типа *VEHICLE*, как в вызове:

fleet.end (*my_vehicle*)

Но полиморфизм играет свою роль, и любой тип, являющийся потомком *VEHICLE*, может прекрасно использоваться. Поэтому наряду с возможностью применения my_vehicle вполне корректно использовать вызов:

```
fleet.extend (cab_at_corner)
```

В общем случае, аргумент может быть любого типа, являющегося потомком *VEHICLE*, таким как *TAXI, TRAM* и другие.

Полиморфный контейнер является результатом последовательности подобных вставок с возможностью различных фактических типов в каждом случае. После нескольких вызовов *extend* наш список *fleet* может выглядеть, например, так:

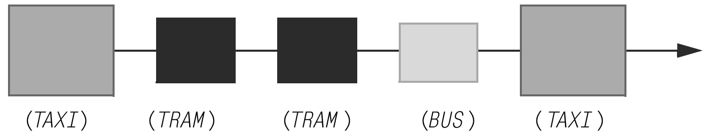


Рис. 16.6. Полиморфный список

Список содержит смесь объектов различных типов, все из них являются потомками *VEHICLE* (включая BUS, не появлявшийся ранее).

Возможность построения таких полиморфных структур данных – результат комбинации двух фундаментальных ОО-механизмов, наследования и универсальности. Это дает нам новый уровень гибкости. Рассмотрим, например, запрос *last*, результатом которого будет последний элемент списка. Он объявлен в классе *LIST*[*G*] и возвращает результат типа *G*. Сущность *fleet* объявлена как *fleet: LIST*[*VEHICLE*], используя *VEHICLE* в качестве фактического родового параметра для *G*. Поэтому выражение:

```
fleet.last
```

будет иметь тип *VEHICLE*. В каждом конкретном случае результирующий объект может быть объектом любого из потомков. Если список находится в состоянии, показанном на последнем рисунке, объектом будет *TAXI*, но это может быть и другой тип, и вы не знаете, какой именно. Но это и не нужно знать, поскольку к результату можно применять любую компоненту класса *VEHICLE*. После объявления *v*: *VEHICLE* и присвоения *v*:= *fleet*. *last* допустимы вызовы *v.load*(...) и *v.count*. Эти вызовы компонентов класса *VEHICLE* корректны, но, конечно же, нельзя вызывать компоненты классов потомков, например, *v.take(...)*, так как *take* ожидает не просто *VEHICLE* аргумент, а *TAXI*.

Я могу услышать от вас: это несправедливо! Просто взгляните на последний рисунок, ведь последний объект – такси. Почему же я не могу выполнить операцию, вполне допустимую для этого объекта?

Не горячитесь.

Во-первых, жизнь полна несправедливостей, и нужно уметь принимать ее такой, какой она есть.

Во-вторых, как вы можете быть уверенным, что возвращенный объект действительно является такси? Рисунок – это просто рисунок, нет никаких гарантий, что при следующем выполнении будет находиться в конце списка *fleet*.

И третье: все будет хорошо – и это настоящий ответ! Существуют способы проверки того, чем является полученный объект, является ли он в самом деле такси в данном конкретном выполнении, и если да, то вызов *take* можно сделать законным. Но прежде чем узнать, как это делается, придется прочесть еще несколько десятков страниц. Разве я не говорил вам, что жизнь полна несправедливостей?

В данный момент более важно понимание эффекта правильных вызовов – вызовов компонентов *VEHICLE*, таких как *v.load*(...) и *v.count* для случая полиморфной цели. Ответ ведет нас к еще одной фундаментальной ОО-концепции.

16.3. Динамическое связывание

В вызове, таком как *v.load*(...), цель *v* является полиморфной, так что во время выполнения она может быть присоединена к объекту типа *TAXI*, или *TRAM*, или к любому потомку *VEHICLE*.

Если теперь вы посмотрите на текст этих классов, то увидите, что каждый из них имеет собственную реализацию метода *load* (ниже мы увидим, как создавать такие объявления без двусмысленностей и конфликтов). Какая же версия должна реально вызываться?

Единственно правильный ответ – мы хотим вызывать *правильный компонент*. Правильный в том смысле, что он наиболее точно соответствует типу объекта, к которому присоединена сущность *v* в момент выполнения. Когда этот объект является экземпляром *TAXI*, мы хотим, чтобы вызывалась версия класса *TAXI*, когда *TRAM* – то *TRAM*-версия, и так далее.

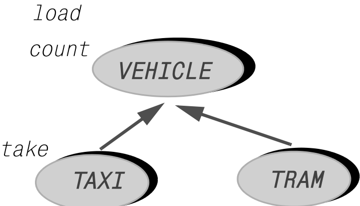


Рис. 16.7. Знакомое наследование

Любое другое решение было бы некорректным. Глупо было бы применять операцию посадки в трамвай при посадке в такси. Даже применять операцию по умолчанию, из класса *VEHICLE*, даже если она там предусмотрена, нецелесообразно. Понятно, что автор класса *TAXI*, который задал реализацию *load*, специально приспособленную для посадки в такси, был бы крайне удивлен, если бы использовалась для этих целей другая реализация, предназначенная для посадки в другое транспортное средство.

Объявление *v: VEHICLE* имеет целью сделать переменную *v* достаточно общей, чтобы при разных выполнениях она могла связываться с разными объектами – иногда с такси, иногда с трамваем, с различными видами транспортных средств. Но в любом конкретном вызове она всегда обозначает не какой-то абстрактный объект – она всегда присоединена к объекту конкретного типа. Компоненты, которые применяются к объекту, должны учитывать его специфику.

Эта политика, являясь краеугольным камнем конструирования ОО ПО, имеет собственное имя.

Определение: динамическое связывание

Динамическое связывание (семантическое правило) требует, чтобы при вызове компонента использовалась та версия компонента, которая наилучшим образом адаптирована к типу целевого объекта.

Противоположная позиция носит название *статического связывания*. Это может звучать удивительно, но статическое связывание применяется во многих языках. Прежде всего стоит упомянуть язык C++ и его потомков, где статическое связывание является политикой, используемой по умолчанию, и применить динамическое связывание возможно только по явному требованию, объявив соответствующий компонент (функцию или метод в C++) виртуальным (**virtual**).

16.4. Типизация и наследование

Как вы, вероятно, уже догадались, есть предел гибкости типизации, обеспечиваемой полиморфизмом. Все хорошо, когда переменная типа *VEHICLE* обозначает объект *TAXI* или *TRAM*, но было бы неправильно позволять ей присоединяться к объектам, представляющим пассажиров или город.

Правило прямолинейно: при полиморфном присоединении тип источника должен быть потомком типа цели. Это не вполне подходящая терминология, – ниже мы увидим, как сделать ее корректной, – но она выражает основную идею. Отсюда следует, что наш прежний

пример присваивания *my_vehicle:= cab_at_corner* является корректным, а присваивание *my_vehicle:= Paris* – недопустимо. Город Париж не является транспортным средством.

Это правило стоит за разнообразием возможностей, обсуждаемых ранее. Если мы допускаем присваивание *x:= y*, где *x* типа *T*, а *y* типа *U*, мы должны быть уверенными, что вызов *x.f* будет иметь смысл во время выполнения, не только если целевой объект типа *T*, но и в том случае, если он имеет тип *U*, при условии, что вызов признан корректным во время компиляции. Это условие корректности является обычным и основывается на объявлении *x*, оно устанавливает, что *f* должно быть компонентом *T*. Но нам нужны гарантии, что *f* также является компонентом *U*. Это выполняется по определению, если *U* потомок *T*.

Следующие термины помогают в понимании этих концепций.

Статический тип, динамический тип

> **Статическим типом** сущности или выражения *e* является тип, используемый в объявлении в соответствующем тексте класса.
> Если *e* во время конкретного выполнения присоединено к объекту, то тип этого объекта будет **динамическим типом** *e*.

После присваивания *my_vehicle:= cab_at_corner* сущность *my_vehicle* имеет динамический тип *TAXI*. Ее статический тип является типом, заданным в объявлении – *VEHICLE*.

Эти понятия применимы к сущностям и выражениям. Для *объектов* такого различия типов нет. Для них определен только динамический тип, появляющийся в момент создания объекта, — это тип *TAXI*, или *TRAM*, или что-либо еще, но всегда остающийся неизменным конкретный тип объекта.

Следствием этого базисного правила для типов является утверждение, что **любой динамический тип сущности или выражения должен быть согласован со статическим типом**.

Используемый здесь термин «*согласование*» распространяет на типы известное нам требование «быть потомком» для классов. Как вы помните, различие между типами и классами связано с универсальностью. Класс, не являющийся родовым, задает тип, (что позволяло нам обходиться до сих пор в нашем обсуждении понятием «потомок»). Но если класс универсальный и имеет родовые параметры, то нужно обеспечить фактические родовые параметры, которые сами являются типами, чтобы превратить класс в тип. *Согласование* имеет место для ссылочных типов, если между лежащими в основе классами имеет место *потомственность* и совпадают фактические родовые параметры. Вот поясняющие примеры:

- *TAXI* (не имеет родовых параметров) *согласовано* с *VEHICLE,* так как является его потомком;
- *LINKED_LIST*[*TAXI*] *согласовано с LIST*[*TAXI*], так как *LINKED_LIST* является потомком *LIST* и фактический родовой параметр один и тот же.

Дадим более точное определение этого свойства.

Определение: согласование

> Если класс *D* является потомком класса *C* и оба не развернутые, то типы, выводимые из *D*, *согласованы* с теми, что выводимы из *C*, следующим образом:
> - если класс *D* не являются универсальным, то *D* (как тип) согласован с *C*;
> - если класс *D* является универсальным, то *D*[*T*, *U*, …] согласован с *C*[*T*, *U*, …] (с теми же самыми родовыми параметрами).
>
> Развернутый тип согласован только с собой.

Полное определение допускает соответствие и в том случае, когда фактические родовые параметры не совпадают, но согласованы друг с другом. Так, *LINKED_LIST*[*TAXI*] согласован с *LINKED_LIST*[*VEHICLE*] (следовательно, и с *LIST*[*VEHICLE*]), поскольку *LINKED_LIST* согласован с *LIST*, а класс *TAXI* согласован с *VEHICLE*. Этот случай требует особой внимательности, и нам он не нужен. Как обычно, когда возникает какая-либо неясность, следует обращаться к стандарту языка для полного понимания.

Все это нам понадобилось, чтобы дать корректную версию правила полиморфизма. Точное правило будет ссылаться на «согласование», а не на свойство «является потомком».

Правило полиморфизма типов

Чтобы полиморфное присоединение было правильным, тип источника должен быть согласован с типом цели.

Это подтверждает, что полиморфизм применим только к ссылочным типам, так как по определению развернутый тип согласован только с самим собой.

Правило типизации обеспечивает нам безопасность, гарантируя, что в фундаментальной операции ОО-программирования

```
x.f (…)
```

всегда будет компонент *f*, применимый к объекту, к которому присоединен *x* во время выполнения, даже если его тип может быть результатом полиморфного присоединения к *x*.

Интересно отметить комбинацию свойств, которые характерны для большинства современных ОО языков программирования.

- **Статическая типизация**, гарантирующая, что существует *по меньшей мере один* компонент для *f*.
- **Динамическое связывание**, гарантирующее использование *лучшего* компонента – того, что непосредственно подходит типу объекта, если доступно более одного варианта реализации *f*.

Язык Smalltalk выбирает из этой политики комбинацию динамического связывания и динамической проверки типов. Как результат, неверное применение вызова компонента, например, *Paris*. *load*(...) (компонент класса *VEHICLE* применяется к цели класса *CITY*), не будет обнаружено на этапе трансляции, и ошибка проявится только во время выполнения, приводя к аварийному завершению программы. Цель такого подхода в том, чтобы избежать излишних проверок в период компиляции и обеспечить большую гибкость.

Другая группа ОО-языков в целях улучшения производительности имеет тенденцию применять статическое связывание.

16.5. Отложенные классы и компоненты

Взгляните на список компонентов класса *VEHICLE* (рис 16.8).

Здесь можно увидеть компонент *move_next*. В тексте класса он не появляется, так как наследуется от класса *MOVING*. Но в тексте *MOVING* вы найдете следующее объявление

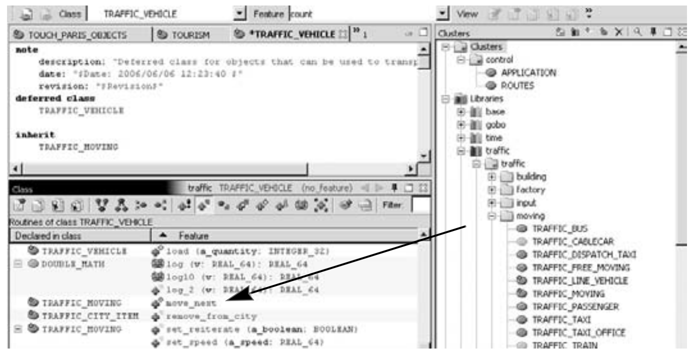


Рис. 16.8. Компоненты VEHICLE в EiffelStudio

```
move_next
            -- Переместиться в следующую позицию, согласно расписанию.
    deferred
    end
```

Это новая форма объявления. Ранее объявление метода имело тело, начинающееся с ключевого слова **do** с последующими за ним операторами. Объявление компонента как **deferred** означает, что компонент имеет спецификацию – сигнатуру и контракт, но не имеет реализации. Реализация отложена (отсюда и имя ключевого слова) и возлагается на классы-потомки. Действительно, в классе *TAXI* можно увидеть реализацию:

```
move_next
            -- Переместиться в следующую позицию, согласно расписанию.
    do
            … Последовательность операторов…
    end
```

Реализация учитывает специфику объектов класса *TAXI*. Класс *TRAM* обеспечивает свою специфическую реализацию компонента.

Мы говорим, что эти классы делают компонент **эффективным**. Отныне мы будем иметь два вида компонентов класса: эффективные (спецификация + реализация) и отложенные (только спецификация). От компонентов эти термины переходят на классы и их типы.

Отложенные классы и типы

Если класс *C* имеет, по крайней мере, один **отложенный компонент** (либо введенный в классе, но объявленный как отложенный, либо наследуемый отложенный компонент, не ставший эффективным), то сам класс считается **отложенным классом**.

Тип, базирующийся на отложенном классе, является **отложенным типом**.

Тип или класс является **эффективным**, если он **не отложен** (все его компоненты эффективны).

Как обычно, различие между классами и типами связано с универсальностью. Вы могли видеть, что класс *LIST* является отложенным классом (поскольку он описывает общее понятие списка с реализацией, возложенной на эффективных потомков, таких как *LINKED_LIST*). Типы *LIST*[*TAXI*] и более общий тип *LIST*[*G*] для любого *G* являются отложенными типами.

Причина, по которой *move_next* является отложенным компонентом класса *MOVING*, состоит в том, что на этом уровне:

- мы знаем, что компонент применим ко всем движущимся объектам, и хотим быть уверенными, что все фактические объекты будут обладать некоторой его версией;
- поскольку *MOVING* задает общую концепцию движущихся объектов, а не специальный вид, то на этом уровне невозможно обеспечить приемлемую реализацию по умолчанию.

Класс *MOVING* специфицирует существование компонента, но ответственность за подходящую реализацию лежит на его правильных потомках, таких как *TAXI*.

Отсутствие реализации по умолчанию в классе *MOVING* создает проблему: каков будет эффект от вызова *m. move_next,* если объект, присоединенный к *m*, имеет тип *MOVING* или *VEHICLE,* когда в обоих классах компонент отложен и, следовательно, не имеет реализации? Ответ прост: таких объектов не существует.

Правило создания «Нет отложенному типу»

Целевой тип в операторе создания не может быть отложенным.

Это защищает нас от создания объектов типа *MOVING* или *VEHICLE* (например, при попытке **create** *my_vehicle*). Благодаря этому становится невозможным вызывать отложенный метод, такой как *move_next* объектом *my_vehicle*.

Так что, если тип отложен, то нет объектов этого типа. Но при этом можно иметь *переменные* (и другие сущности и выражения) этого типа, например, *my_vehicle*. Эти переменные можно присоединять к эффективным объектам соответствующих типов, таких как *TAXI* в нашем примере. Фактически вся идея отложенных компонентов, классов и типов имеет смысл только благодаря полиморфизму и динамическому связыванию.

Поскольку создание требует эффективного типа, при желании создать объект необходимо знать, эффективным или отложенным является соответствующий класс. Чтобы выяснить это, недостаточно знакомства с методами класса, поскольку отложенные компоненты могут наследоваться. Вообще изучение текста классов противоречит принципу скрытия информации. «Быть отложенным» – это ключевое свойство класса, явно им провозглашаемое. Фактически, сразу же после раздела **note**-класса, поставляющего общую информацию о классе, первое, что говорит класс всему миру, – является ли он отложенным.

Правило отложенного класса

Объявление отложенного класса должно начинаться с ключевых слов **deferred class** (вместо просто **class** для эффективных классов).

С точностью до синтаксиса и терминологии предыдущие правила не зависели от языка программирования. Данное же правило является правилом языка Eiffel.

В данном правиле «отложенный класс» понимается в соответствии с определением как класс, у которого, по крайней мере, один из методов не имеет реализации и, следовательно,

является отложенным. При этом метод может быть как наследуемым, так и введенным в классе. Правило гласит, что это знание не должно оставаться информацией для внутреннего употребления, но должно стать частью интерфейса класса, отражаемого в контрактном облике класса. В нашем примере можно проверить, что классы *MOVING* и *VEHICLE* начинаются с **deferred class**.

В языке Eiffel разрешается объявлять класс как **deferred class** даже в том случае, когда у класса нет отложенных компонентов. Такой класс также рассматривается как отложенный, расширяя наше определение. Это полезно, если вы проектируете класс как предка семейства классов и хотите избежать создания экземпляров этого класса.

Будем более точными: объявление класса **deferred** защищает от создания прямых экземпляров. Концепции этой главы требуют пересмотра понятия «экземпляр». До сих пор экземпляр типа (или лежащего в его основе класса) понимался просто как объект периода выполнения, соответствующий статическому описанию, задаваемому классом. Все это правильно, но теперь требуется ввести уточнения и различать прямой и непрямой случаи.

Экземпляр, прямой экземпляр

Объект, полученный в результате выполнения оператора создания, является **прямым экземпляром** целевого типа этого оператора.
Экземпляром типа является **прямой экземпляр** любого согласованного типа.

Следствием правила создания «Нет отложенному типу» является то, что у отложенного типа нет *прямых* экземпляров, как в случае с классами *MOVING* и *VEHICLE*. Но оба класса могут иметь экземпляры – прямые экземпляры эффективных потомков, таких, как *TAXI*.

Эти определения непосредственно отражают концепцию полиморфизма: объявление *x* типа *T* означает, что вы хотите, чтобы во время выполнения *x* обозначал экземпляр этого типа. С введением полиморфизма мы полагаем, что теперь понимаются не только объекты, непосредственно выводимые из *T* – прямые экземпляры, но и, рекурсивно, экземпляры любого согласованного типа (потомков *T*).

Отложенные классы являются главной составляющей эффективности наследования как механизма таксономии. Часто на высших уровнях иерархии наследования находятся главным образом отложенные классы, задающие абстрактные концепции, чьи реализации возлагаются на эффективных потомков. Полезно познакомиться с двумя подробно разработанными примерами.

- Библиотека EiffelBase, содержащая реализацию структур данных и алгоритмов. Библиотека вводит глобальную таксономию фундаментальных структур, используемых в информатике. На вершине иерархии можно видеть такие классы, как *LINEAR* (структуры, обходимые линейно) и *FINITE* (конечные структуры). Ниже по иерархии находятся классы, задающие специфические структуры, которые являются эффективными потомками классов верхнего уровня.
- Графическая библиотека EiffelVision. Здесь также существует кластер классов, задающих геометрические фигуры, на верхнем этаже которого находится отложенный класс *FIGURE*, а ниже по иерархии находятся классы, задающие такие фигуры, как круг и квадрат. Таксономия, принятая для геометрических фигур, служит излюбленным примером в учебниках, рассматривающих наследование. Но в данном случае здесь нет ничего академического – это полезная часть важной прикладной библиотеки классов.

Некоторые языки программирования, в том числе такие известные, как C# и Java, предлагают языковую конструкцию, называемую interface (это понятие рассматривается в при-

ложениях, посвященных этим языкам). Класс, объявленный с таким ключевым словом, называется интерфейсом, и его главная особенность состоит в том, что все его методы отложены (не имеют контрактов) и должны быть реализованы потомками интерфейса[1].

Отложенные классы являются более общей концепцией: как следует из определения, класс отложен, как только он имеет *один* отложенный метод, но он может иметь смесь отложенных и эффективных методов. Это дает возможность использования механизма отложенных классов для возрастающего проектирования. Проектирование обычно начинается с задания абстракций, которые определяются в виде классов, в которых большинство методов отложено (по духу близкие к интерфейсам, где все методы отложены). Затем по мере уточнения проблем, появляются потомки, реализующие часть отложенных методов. На заключительных этапах проектирования строятся полностью эффективные классы. Это и есть ОО-вариант общего процесса уточнения в процессе проектирования.

Мощным приемом, включающим частично отложенные классы, является образец проектирования, называемый «*Программа с дырами*», в котором эффективные методы вызывают отложенные. Такой метод описывает некоторую общую схему решения задачи, оставляя часть деталей для реализации потомкам. Этот подход позволяет комбинировать повторное использование и адаптивность к конкретным условиям. Типичным примером, встречавшимся многократно, служит стандартная схема повторения (итерирования), где некоторая операция выполняется над всеми элементами последовательной структуры, подобной списку. Если ее применить к операции поиска, то она выглядит так:

```
search (v: G)
    — Передвинуться к первой позиции (или после текущей), где появляется v.
    — Если такой позиции нет, то убедиться, что переменная exhausted будет
    — иметь значение true.
    do
            from until v = item loop forth end
    end
```

Классы EiffelBase представляют списки и другие подобные им структуры как потомков класса *LINEAR*, где и появляется приведенный код (в более полной версии учитывается различие между равенством ссылок и равенством объектов). Реализация *forth,* однако, зависит от реализации, выбранной для списков (использующей массив, ссылочные структуры и так далее). Поэтому в классе *LINEAR* метод *forth* является отложенным:

```
forth
            — Переместить курсор к следующей позиции
    require
            in_range: not after
    deferred
    ensure
            increased: index = old index + 1
    end
```

[1] Наряду с интерфейсами в языке C# существует понятие абстрактного класса, совпадающего по концепции с понятием отложенного класса и использующегося в тех же целях, что и отложенный класс. Интерфейсы C# используются в интересах множественного наследования.

Обратите внимание на предусловие и постусловие: как обсуждается ниже, контракты полностью применимы к отложенным компонентам и классам.

Каждый эффективный потомок *LINEAR* реализует *forth*. Для списка, построенного на массиве, где курсор задается индексом, достаточно выполнить присваивание *index*:= *index* +1. Для связного списка (*LINKED_LIST, TWO_WAY_LIST*) детали более сложны. Но методу *search* нет дела до деталей: все, что ему нужно, – это вызвать *forth*, зная спецификацию, заданную контрактом, а не реализацией.

Имя «*Программа с дырами*» отражает подход к возрастающему конструированию ПО. На каждом уровне абстракции используется вся известная на этом уровне информация: та, что может быть полностью реализована (эффективные компоненты, не включающие вызовов отложенных методов), те методы, которые могут быть только специфицированы, и те, которые допускают реализацию, использующую вызовы отложенных методов, подобные *search* в *LINEAR*. Мы можем рассматривать результат как частично сконструированную программу, *дыры* которой должны быть заполнены в процессе уточнения при конструировании потомков.

Этот подход – один из наиболее важных вкладов ОО-методологии в упорядоченное конструирование систем, особенно больших систем.

16.6. Переопределение

Когда класс реализует отложенный компонент, он задает первую реализацию метода, который до сих пор у предков имел только спецификацию. Чтобы использовать в дальнейшем преимущества динамического связывания и сделать архитектуру еще более гибкой, разрешается потомку дать свою собственную реализацию, несмотря на то, что родитель *уже задал* родительскую реализацию компонента. Мы будем говорить, что класс **переопределяет** (**redifine** в Eiffel, **override** в C++, C#) компонент. Переопределение дополняет только что изученный механизм эффективизации:

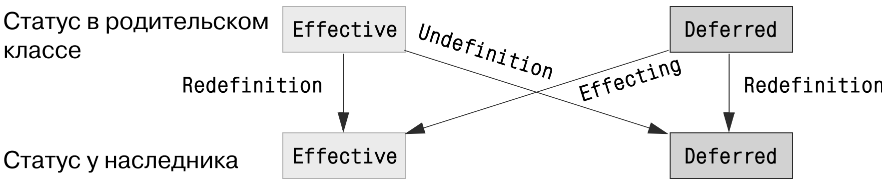


Рис. 16.9. Формы переопределения

На диаграмме наиболее правый случай показывает, что при наследовании отложенного компонента можно не только сделать его эффективным (следуя ветви, помеченной как Effecting), но также изменить его, оставляя на время отложенным. Это происходит в том случае, когда наследник меняет спецификацию метода – сигнатуру или контракт, – что также рассматривается как переопределение. В дополнение к списку вариантов отметим возможность *отмены реализации* (Undefinition), при которой реализация родителя забывается, метод снова становится отложенным и готов начать новую жизнь.

Термин *переобъявление* покрывает все эти случаи.

Определение: переобъявление (redeclaration)

Переобъявление наследуемого компонента означает изменение чего-либо или всего – сигнатуры, контракта, реализации, а также удаление реализации. Варианты включают задание реализации (*effecting*), переопределение (*redefinition*), отмену реализации (*undefinition*).

Нет необходимости в изучении отмены реализации, поскольку это своеобразная операция (хотя в ней ничего трудного или загадочного – примеры ее применения можно найти в EiffelBase, проведя поиск по ключевому слову **undefine**). Переопределение, с другой стороны, – широко распространенная операция.

В качестве примера рассмотрим из библиотеки *Traffic* класс *DISPATCH_TAXI*, который наследует от *TAXI* и представляет понятие такси, находящегося под управлением диспетчерской службы, в противоположность тем такси, которые курсируют по улицам и ищут пассажиров, полагаясь на собственную удачу. Процедура *take* по-разному реализована для этих двух типов такси, поскольку в случае диспетчера необходимо уведомлять его о каждой посадке и высадке пассажиров. На диаграмме наследования переопределяемый компонент отмечен как «++» (идея в том, что определение дает один +, а второй добавляется при изменении реализации, делая метод «более эффективным»).

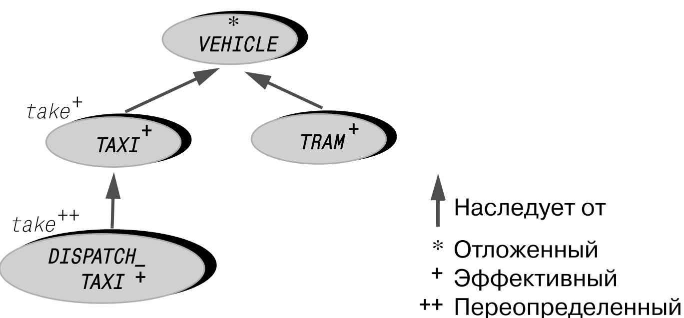


Рис. 16.10. Переопределение

Отсутствие любого символа означает по умолчанию «эффективный». На рисунках обычно один + опускается, за исключением тех случаев, когда необходимо подчеркнуть, что имеет место эффективность.

Для переопределения наследуемого компонента необходимо задать его новое объявление в тексте класса, но также необходимо объявить ясно, громко и недвусмысленно всему миру о таком переопределении. Для этого задается предложение **redifine** в соответствующей **inherit** части класса:

```
class
    DISPATCH_TAXI
inherit
    TAXI redefine take end
feature
```

```
    take (from_location,to_location: LOCATION)
            – Перевезти пассажиров из from_location в to_location
            do
                    … Новая реализация…
            end
    … Другие компоненты и оставшаяся часть класса …
end
```

Если класс переопределяет несколько компонентов, он перечисляет все: **redifine** *f, g*, …**end**

Цель предложения **redefine** – это ясность и безопасность. Важное правило надежного ОО-программирования состоит в том, что имя компонента в классе не должно использоваться для именования двух различных методов. Эта ситуация, известная как перегрузка (overloading), допускается в некоторых языках программирования, что увеличивает риск непонимания. Предложение **redefine** поясняет клиенту класса, читающему текст, отношение с компонентом родителя, имеющим совпадающее имя: это не новый компонент с тем же именем, а переопределение компонента.

Если пропустить предложение, возникнет ошибка на этапе компиляции, так как класс полагает, что в нем определены два метода с одним именем (перегрузка), что запрещено правилами Eiffel.

Требование перечисления наследуемых компонентов в **redefine** применимо только к переопределению. При отмене реализации необходимо аналогичным образом использовать предложение **undefine,** обеспечив в то же время новое объявление метода как отложенного. При задании реализации отложенного метода никаких специальных предложений не требуется. Конфликты при этом не возникают, новый эффективный компонент естественно включается в наследуемую версию, поставляя долгожданную предками реализацию.

Переопределение, подобно эффективизации, использует динамическое связывание:

```
group_move (taxi_ fleet: LIST [TAXI]– Смотри далее о классе TARGET
            – Заставляет все такси в taxi_fleet следовать одним маршрутом.
    do
            from taxi_ fleet.start until taxi_ fleet.after loop
                    taxi_ fleet.item.take(…)
                    taxi_ fleet.forth
            end
    end
```

В этой программе каждый элемент списка будет выполнять версию *take* либо из класса *TAXI*, либо *DISPATCH_TAXI*, в зависимости от того, чьим прямым потомком является элемент. Это подобно нашим прежним примерам с полиморфными переменными и структурами данных. Единственная разница в том, что *TAXI* эффективно и, следовательно, имеет прямые экземпляры, в то время как *MOVING* и *VEHICLE* отложены и таковых не имеют.

Если вы переопределяете метод, родительская версия известна как предшественник (precursor) метода. Довольно часто новая реализация основывается на версии предшественника. Вместо простого дублирования кода (разве я не упоминал, что copy-paste – не лучшая идея?) можно использовать ключевое слово **Precursor**. Новая реализация *take* в *DISPATCH_TAXI* выглядит следующим образом:

```
take (from_location,to_location: LOCATION)
            — Перевезти пассажиров из from_location в to_location
    do
            Precursor(from_location, to_location)
            … Другие операции, которые характерны для такси, контролируемых
              диспетчером …
    end
```

Это означает, что вначале выполняется версия, предусмотренная предшественником, а затем добавляются операции, учитывающие специфику.

В переопределяемом методе можно использовать слово **Precursor** в качестве имени родительского метода, передавая при необходимости аргументы.

16.7. За пределами скрытия информации

Комбинация описанных механизмов играет ключевую роль в построении гибких, повторно используемых и (благодаря статической типизации и контрактам) надежных систем. Особый вклад полиморфизма и динамического связывания позволяет сделать еще один шаг на пути скрытия информации. Основная идея скрытия информации состоит в том, чтобы позволить клиентам использовать механизмы поставщиков без знания того, как они реализованы. Следующим шагом является защита клиента от знания того, *какой именно из возможных поставщиков* использовался в данном конкретном случае. Когда вы пишете

```
my_vehicle.load (…)
```

вы запрашиваете некоторую абстрактную операцию *load*, применяемую к целевому объекту, но так как вам не известно (переменная *my_vehicle*, возможно, полиморфна), какой точно тип имеет связанный с ней объект, — динамическое связывание означает, что вы не знаете, какой именно метод будет вызван.

В этом причина важности контрактов. То, что вы должны знать, заключено в исходном контракте *load*, включенном в отложенный класс *VEHICLE*. Контракт описывает семантику *load* — загрузку *n* пассажиров в транспортное средство. Это справедливо для всех вариантов, хотя каждый из них отличается деталями реализации.

Выбор из многих вариантов

Оценивая значимость ОО-стиля, допускающего полиморфизм и динамическое связывание, рассмотрим, что пришлось бы нам делать в случае отсутствия этих механизмов на примере той же задачи адаптации операции к типу ее цели. Конечно, нетрудно построить такое решение:

```
load (v: VEHICLE; n: INTEGER)
            — Загрузить n пассажиров в v.
    do
            if "v объект tram" then
                    "Применить алгоритм посадки в трамвай"
            elseif "v объект taxi" then
                    "Проверить, что число пассажиров не более 4-х"
```

```
                    "Применить алгоритм посадки в такси"
            elseif …
    end
    end
```

Мы просто проверяем тип и в зависимости от типа применяем соответствующий алгоритм. Этот способ хорошо работает, но у него есть неприятные следствия.

- Когда вариантов много, приходится строить длинный, а в результате сложный оператор **if** (применение конструкции с разбором случаев в таких ситуациях гораздо предпочтительнее).
- Приходится повторять вызов каждой операции, такой как *load*, которая концептуально применима к любому транспортному средству, хотя выполняется по-разному. Таких операций может быть много.

Проблема с таким многословным и повторяющимся кодом в том, что он препятствует процессу гладкой эволюции ПО. Необходимость добавления нового типа – нового транспортного средства в системе Traffic – возникает достаточно часто. Тогда приходится искать все программы, выполняющие проверку типов, аналогично приведенному примеру, изменять их текст, включая вновь появившийся тип.

В противоположность этому динамически связываемое решение требует только добавления класса, привязки его к нужному месту иерархии наследования, реализации методов, специфических для этого класса. Чаще всего не приходится изменять уже существующее ПО.

Появление в ПО структуры решения с многими ветвями, как в вышеприведенной программе, *должно настораживать*, поскольку может быть результатом плохого проектирования.

Почувствуй методологию

Сражение с синдромом «Много явных вариантов»
Если ваше решение приводит к построению структуры со многими ветвями, проверьте, не проще ли построить решение, основанное на динамическом связывании.

Этот совет проектировщику не является абсолютным правилом. Не все условные операторы следует заменять динамическим связыванием. Наиболее «подозрительными» кандидатами являются проверки типа объекта, как в примере. Вместо алгоритма, осуществляющего выбор между возможными типами объектов и, следовательно, обязанного знать все возможные типы, лучше выбрать алгоритм, основанный на наследовании и динамическом связывании.

Наследование делает такое решение лучшим. При добавлении класса, как только найдено для него соответствующее место в иерархии наследования, все, что вам потребуется, это переобъявление методов, специфических для этого класса. Все остальное будет наследовано.

Это обсуждение предполагает, что мы имеем дело с новыми типами, появляющимися в процессе эволюции, для которых известно множество операций. Советы не применимы к обращенной ситуации, когда добавляются как типы, так и операции. Образец «Visitor» (посетитель), рассматриваемый в конце этой главы, дополняет обсуждение.

16.8. Беглый взгляд на реализацию

В целом эта книга представляет концепции программирования с позиций их использования прикладными программистами (но не разработчиками компиляторов – тема замечательная,

но для другой книги). Мы уже сделали несколько исключений; реализация динамического связывания станет еще одним. Так что на время прервемся, уйдем из области чистых концепций, откроем капот и займемся мотором – рассмотрим реализацию, что даст нам лучшее понимание основ проблемы.

Для простоты будем рассматривать вызовы методов (хотя ситуация с атрибутами подобна). Без динамического связывания компилятор, когда он видит вызов, знает, какой метод следует вызывать в генерируемом коде. При статическом связывании для вызова

```
cab_at_corner.load (…)
```

должна применяться версия метода *load* класса *TAXI*, что выводимо из типа, заданного при объявлении цели *cab_at_corner*. Для описания генерируемого кода позвольте использовать язык *C*. Он является языком достаточно низкого уровня и может служить представителем ассемблерных языков, но все же он достаточно высокого уровня, чтобы оставаться понимаемым и не зависимым от конкретной платформы (помимо всего, компилятор EiffelStudio в качестве одного из своих возможных выходов генерирует код на языке *C*, так что рассматриваемая ситуация реалистична).

Без динамического связывания генерируемый код для вышеприведенного вызова будет выглядеть примерно так:

```
C_TAXI_load (C_cab. …);                                    [C1]
```

Здесь *C_TAXI_load* является результатом трансляции вызова метода *load* для версии класса *TAXI*. Так как *C* не является ОО-языком и не имеет понятия квалифицированного вызова *x.f*(…), функции в языке *C* имеют, по крайней мере, один аргумент, соответствующий цели, – здесь *C_ cab* представляет исходный *cab_at_corner* в Eiffel.

В этом статическом варианте программа *C_TAXI_load* известна на этапе компиляции, и она может появиться в генерируемом коде явно под своим собственным именем.

При динамическом связывании такая ситуация более не происходит. На этапе компиляции доступны различные версии *load*, и только на этапе выполнения становится ясным, какая из этих версий должна выполняться в текущей точке вызова. Генерируемый код

```
v.load (…)
```

должен основываться на некоторой подходящей структуре данных, подобной той, что показана на рисунке.

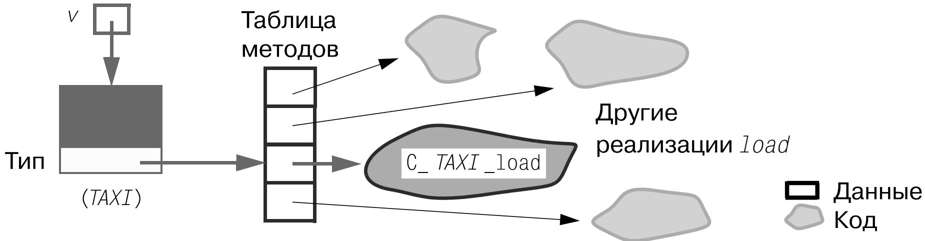


Рис. 16.11. Разрешение динамически связываемого вызова

Ключевой структурой является таблица методов[1] (называемая также таблицей переходов). Входами в этой таблице являются ссылки. В отличие от обычных ссылок они указывают не на данные, а на код – в данном примере на соответствующую версию метода *load*.

Такие ссылки (или «указатели») на код не есть нечто такое, что можно непосредственно получить в Eiffel или в большинстве ОО-языков высокого уровня, но на машинном уровне они являются прямым результатом концепции «программы, хранимой в памяти». Так как код, подобно данным, хранится в памяти, то вход в таблице может содержать адрес начала блока кода. В дополнение к этому, множество команд компьютера включает команду в форме: «выполнить код, начинающийся с адреса *addr*».

Языкам высокого уровня нет необходимости в таком свойстве, поскольку оно чревато ошибками: что, если по адресу *addr* хранится не код, а данные, или, еще хуже, код, размещенный зловредным хакером? Для ОО-языков динамическое связывание является безопасной заменой. В Eiffel имеется еще один подобный механизм – агенты. Агент можно рассматривать как некоторую обертку метода, и он может быть передан различным частям ПО, позволяя им вызывать метод. В некоторых не ОО-языках предоставляется возможность передавать метод в качестве фактического аргумента другим методам. В главе, посвященной агентам, обсуждаются эти механизмы, позволяющие отложить решение по выбору метода до момента его исполнения.

При динамическом связывании выбор метода зависит от типа целевого объекта. На последнем рисунке этот объект, показанный слева, является экземпляром *TAXI*. Мы не можем вывести это, анализируя текст программы, где известно только, что переменная *v* относится к более общему типу *VEHICLE*; но если *v* полиморфна, то соответствующая ссылка может быть в некотором сеансе выполнения присоединена к экземпляру *TAXI*. Вся идея динамического связывания и состоит в том, что объявление типа *v* означает лишь, что при сохранении общности допускаются разные типы в разных сеансах выполнения.

Если рассматривать эти свойства с позиций разработчика компилятора, то приходим к важному заключению – **каждый объект должен содержать идентификацию своего собственного типа**. В противном случае не было бы возможности добиться динамического связывания, так как именно тип определяет, какой вариант метода следует выбрать из множества возможных. Реализации ОО-языков действительно включают в представление любого объекта в дополнение к его полям, задающим атрибуты, поле, задающее тип (на рисунке оно помечено как «Тип»). Обычно тип задается целым числом и для него отводится одно слово памяти (4 или 8 байтов), этого достаточно в современных прикладных системах для задания любого возможного типа. Реализация EiffelStudio добавляет все же еще одно слово к каждому объекту для контроля информации, необходимой, в частности, сборщику мусора. Таковы издержки ОО-механизмов в этой реализации – два дополнительных слова на каждый объект. В целом это приемлемая плата, но могут возникать проблемы, если создается очень много небольших объектов.

Можно пройти по пути, показанному на рисунке, начав с объекта *v* слева вверху, следуя стрелкам. Если значение *v* не равно void, то ссылка приведет нас к объекту. Поле «Тип» этого объекта даст нам соответствующее целое, представляющее тип объекта (здесь *TAXI*). Мы используем это целое для индексации входа в таблицу методов, построенную для *load*; соответствующий вход даст адрес программного кода соответствующего варианта метода.

[1] При реализации языков, подобных C++, C#, где динамическое связывание применяется к методам, объявленным как виртуальные, такая таблица называется таблицей виртуальных методов

Буквально, эта схема даст нам следующий *C*-код, который можно сравнить с кодом, полученным для случая статического связывания:

```
(routine_table[(_v).type]) (v, …);                                        [C2]
```

Пояснение: символ * означает операцию разыменования, так что * *v* – это объект, заданный ссылкой *v*, тогда (**v*).*type* (что может быть также записано как *v*-> *type*) является соответствующим полем объекта и его значение используется как индекс в массиве *routine_table*. Так мы получаем адрес нужной *C*-программы, которой и передаются необходимые ей аргументы. Как и ранее, список аргументов включает помимо исходных аргументов и аргумент, задающий целевой объект, известный здесь как *v*, играющий роль объекта *C_cab*.

Из этой базисной схемы можно вывести много различных вариантов реализации, применяя различные оптимизации. Прежде чем понять всю проблему в целом, заметим, что, поскольку мы имеем дело не только с одним методом *load*, коллекция всех таблиц разных методов представляет двумерную структуру с *T* строчками и *R* столбцами, *T* – задает типы, а *R* – программы методов, как показано на следующем рисунке:

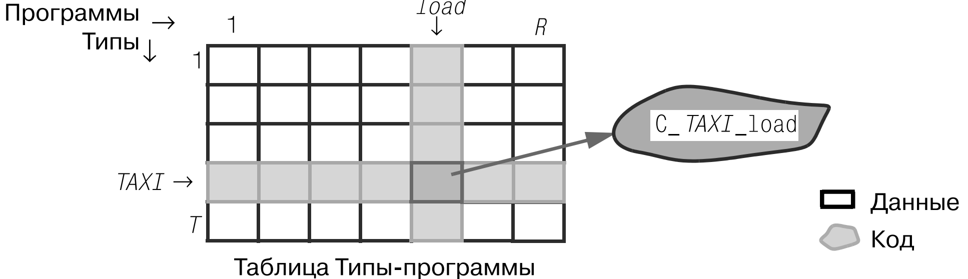


Рис. 16.12. Общая схема реализации динамического связывания

Все эти решения удовлетворяет важному требованию.

Эффективная реализация динамического связывания

> Хорошая реализация динамического связывания должна обеспечивать время поиска нужного варианта за время O(1).

Издержки времени на вызов динамически связываемого метода в сравнении с вызовом статически связываемого метода сводятся к стоимости поиска нужного метода. Гибкость, обеспечиваемая полиморфизмом и динамическим связыванием, имеет свою цену, однако любую цену мы платить не готовы. Но, как только что показано, принципиально возможно выполнять операцию поиска за константное время независимо от числа типов и числа методов в классе.

Наивная реализация сохраняет в памяти структуру, задающую иерархию наследования, и выполняет обход этой структуры в процессе поиска метода. Такая техника *не приемлема* (она особенно плоха в случае множественного наследования). Все обсуждаемые реализации построены на массивах (массив, индексируемый типом, программой, двумерный массив). Все они обеспечивают требуемое время поиска.

Следует учитывать и другую сторону эффективности программы – стоимость требуемой памяти. Во всех трех вариантах потребуются структуры, имеющие в общей сложности $T * R$ входов. Этому требованию иногда трудно удовлетворить. Например, EiffelStudio использует примерно 6000 типов и 50000 методов. Но таблица, показанная выше, избыточна, поскольку практически большинство входов будут пусты, – каждый метод имеет смысл только для нескольких типов, например, метод *load* применим только для *VEHICLE* и его потомков.

Считать нужно программы, а не имена программ, поскольку в разных классах могут существовать программы с одним и тем же именем.

В варианте, который использует таблицы методов, мы можем отрезать края каждой из этих таблиц, удалив в каждом столбце все входы перед первым непустым входом («эффективное начало») и после последнего не пустого входа.

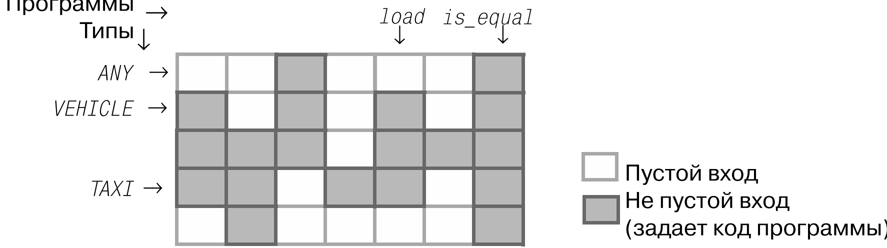


Рис. 16.13. Общая схема реализации динамического связывания

Вышеприведенный код (**C2**) все еще будет работать при условии, что вход в таблицу *routine_table*[*i*] индексируется по отношению к эффективному началу, а не относительно физической точки начала столбца. Достичь этого нетрудно.

Такая оптимизация не представляет особого интереса, если непустые входы распределены по всему столбцу, задающему таблицу методов. Предположим, что при наличии 600 типов номер 1 отведен типу *VEHICLE*, 3000 для *TRAM* и 6000 для *TAXI*. Даже если *load* существует только для этих трех типов, нам все же придется иметь столбец с 600 входами, хотя все из них кроме трех будут пустыми. Для улучшения ситуации заметим, что у нас есть свобода для выбора номеров, присваиваемых типам, так что можно воспользоваться преимуществами следующего свойства.

Теорема о соседстве методов

Классы, обладающие методом *M*, являются потомками класса, где метод *M* впервые появился.

Это предполагает выбор такой схемы нумерации, которая дает соседние номера потомкам любого данного класса. К этому моменту вы уже знаете, что может помочь в данной ситуации, – топологическая сортировка, рассмотренная в предыдущей главе. Отношение «Быть потомком» является отношением частичного порядка, так как наследование ациклично.

Выполнение топологической сортировки существенно уменьшает размер таблиц методов – в EiffelStudio примерно на 85%. Эта техника носит принципиальный характер, поскольку без нее издержки памяти были бы критичными.

Что же касается издержек времени, то, как говорилось, они невелики, требуют константного времени и сравнимы с доступом к полю или элементу массива. Но все же лучше, если можно от них вообще избавиться, и такое в некоторых ситуациях вполне возможно. Компилятор может обнаружить, что:

- некоторая программа имеет только одну версию;
- некоторое выражение не является полиморфным.

В таких ситуациях можно применить схему статического связывания на этапе компиляции **[C1]** и избежать любых потерь времени при выполнении.

По этой причине в ряде языков (C++, Java, C# и других) статическое связывание предполагается по умолчанию, а динамическое – резервируется только для виртуальных методов. Такая политика приводит к появлению проблем, поскольку программисту легко ошибиться, предположив, что вызов является статическим, в то время как он полиморфен. Возможно, что когда-то принятое решение было корректным, но в ходе эволюции добавился новый класс-потомок с новой версией метода, а изменение объявления метода как виртуального так и не было сделано.

Правило говорит, что **динамическое связывание всегда задает корректную семантику** ОО-вызовов. Как следствие, статическое связывание применимо только тогда, когда оно имеет ту же семантику, что и динамическое связывание. Типично это бывает тогда, когда имеет место один из двух вышеописанных случаев. Из-за трудностей обнаружения таких случаев эту задачу целесообразно передать компилятору.

В EiffelStudio компилятор действительно занимается подобной оптимизацией.

Эти наблюдения дополняют наш краткий экскурс в технику реализации. Надеюсь, это дало лучшее понимание значимости наследования и связанных с ним приемов для выполнения ОО-программ. Конечно, на практике приходится учитывать много деталей. Если вы хотите дойти «до сути вещей», то стоит перейти к анализу кода на языке C, генерируемого компилятором EiffelStudio в «классическом» варианте. На следующем шаге следует изучить сам Eiffel-код.

Все доступно, все является открытым кодом. Как итог, два ключевых момента.

- Издержки времени на динамическое связывание могут требовать константного времени и невелики. В ряде случаев применение подходящих приемов может свести их к нулю.
- На реализацию динамического связывания требуется дополнительная память – одно поле на каждый объект, и память для таблиц, которая может быть ограничена до приемлемых размеров, если использовать разумную технику программирования.

16.9. Что происходит с контрактами?

В определение метода входит не только имя, сигнатура и (для эффективных методов) реализация, – определение может также включать предусловие и постусловие. Для класса может быть задан инвариант класса. Мы знаем, что означают эти понятия в отсутствие наследования. А как наследование влияет на эту картину?

Аккумуляция инварианта

Первое правило воздействует на инвариант класса. Оно отражает взгляд на наследование как на отношение «*является*» и на роль наследования как механизма таксономии. Указание того, что класс *TAXI* является наследником класса *VEHICLE*, не только избавляет от дублирования кода, но и задает полиморфизм: когда ожидается транспортное средство *LIST*[*VEHI-*

CLE], то возможно появление такси. Отсюда следует, что любое ограничение, определенное для экземпляров родительского класса, должно применяться и к наследникам.

В классе *VEHICLE* находим:

```
invariant
    not_too_small: count >= 0
    not_too_large: count <= capacity
```

Инвариант выражает тот факт, что число пассажиров транспортного средства не может быть отрицательным числом, и оно не должно превышать *вместимости* этого средства. Этих утверждений нельзя найти в классе *TAXI* не потому, что они перестали быть применимыми, но совершенно по противоположной причине – эти предложения присутствуют здесь автоматически. Класс наследует от родителя не только методы, но и инвариант класса.

Эти наследуемые предложения можно увидеть, просматривая плоский или контрактный облик класса. Понятно, что наследник может вводить дополнительные ограничения. Действительно, в классе *TAXI* можно видеть предложение:

```
invariant
    legal_limit: capacity = 4
    … Другие предложения, которые воздействуют на методы, специфические для такси …
```

Эти утверждения дополняют утверждения родителя. В плоской форме класса вначале перечисляются утверждения родителя, затем потомка.

Возникает естественный вопрос, что происходит в случае возникновения противоречий в утверждениях, например, если потомок устанавливает, что вместимость (*capacity* = -1) отрицательна. Но здесь нет ничего нового, поскольку противоречия возможны и в утверждениях родителя. Такой инвариант просто ошибочен, что и будет незамедлительно обнаружено при тестировании (в будущем при проведении статического анализа).

Следующее определение задает семантику.

Определение: инвариант класса

Инвариантом класса является утверждение (p_1 **and** … **and** p_n) **and then** *i*, где *i* является утверждением, явно заданным в собственном **инварианте** класса (или **True** в случае его отсутствия), а p_1 … p_n являются (рекурсивно) инвариантами родительских классов, если таковые есть.

Определение учитывает возможность существования у класса множества родителей, что будет изучаться позднее в этой главе. Утверждения в **инварианте** класса могут состоять из нескольких подвыражений, как в вышеприведенном примере. В этом случае неявно предполагается, что они соединены связкой **and then**.

Ослабление предусловия и усиление постусловия

Вторая проблема – влияние наследования на предусловие и постусловие – ведет к важному правилу разработки ПО. Для ее понимания следует рассмотреть ее в контексте с полиморфизмом и динамическим связыванием:

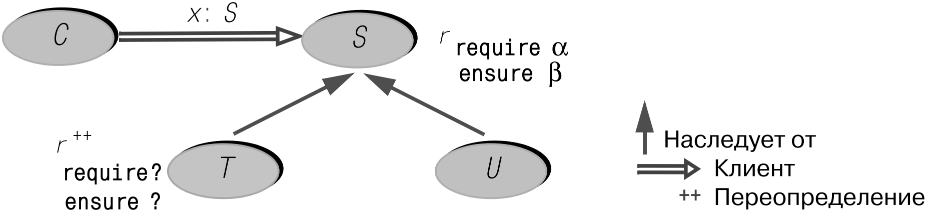


Рис. 16.14. Контекст адаптации контракта

Рассмотрим метод *r* из класса поставщика *S*, заданный с предусловием и постусловием (названными α и β на рисунке). Потомок *T* переопределяет метод *r*, что может быть эффективизацией (заданием реализации), если *r* был отложенным в *S*. Возникает вопрос: какие изменения в контракте (α и β) допустимы для новой версии *r*?

Для получения правильного ответа следует рассмотреть эту ситуацию с позиций класса *C* – клиента класса *S*, в методах которого встречается вызов *x*. *r* (...), где *x* по объявлению имеет тип *S*. Контракт устанавливает права и обязанности клиента: он должен перед вызовом метода гарантировать выполнение предусловия, в этом случае по завершении метода ему гарантируется выполнение постусловия. Возможна, например, такая схема работы клиента:

```
if x._ then
    x.r (…)                                                    [3]
    – Здесь гарантируется, что выполняется x.β
end
```

Это прямое применение принципа проектирования по контракту. Но теперь включается полиморфизм. Наш *x*, типа *S* по объявлению, во время выполнения не обязан быть присоединенным к прямому экземпляру класса *S*, он может обозначать объект класса *T* или экземпляр любого другого класса потомка *S*.

Из-за динамического связывания будет вызвана версия *r*, переопределенная потомком. Но, конечно же, клиенту нет необходимости знать это – он заключил контракт с классом *S*, более того, во время написания клиентского класса *C* класс *T* мог вообще не существовать. Приведенный выше код мог быть частью кода такого метода класса *C:*

```
do_something_with_an_S_object (x: S)
```

Как видите, в данном случае *x* – это аргумент метода, имеющий тип *S*. Фактический аргумент, приходит в класс *C*, возможно, из внешнего мира, и его тип должен быть лишь *согласован* с *S*, так что сам класс *C* изначально может находиться в неведении о фактическом типе *x* в момент вызова. Класс *T* мог быть написан и добавлен в иерархию наследования спустя два года после создания класса *C*. Некий другой программист, знающий о появлении класса *T*, мог написать в своей программе: *c1.do_something_with_an_S_object*(*t1*), где *c1* – класса *C,* а *t1* – класса *T*. Можно только пожалеть автора исходного кода класса *C*, который должен написать клиентский код и гарантировать его корректность даже в том случае, когда он имеет дело с объектами, не существовавшими в момент написания кода.

Для обеспечения корректности *C* может опираться на известные ему свойства поставщиков, таких как *S*, и их методов, таких как *r*. Это строго ограничивает потомков, например *T*, не позволяя им «баловаться» с контрактом, допуская, например, такие вольности:

- усиливать предусловие *r* в *T*. В этом случае вызов, такой как [3], уже не гарантировал бы нормального выполнения, поскольку клиент гарантирует выполнения условия α, а этого становится недостаточно для объектов типа *T*;
- ослаблять постусловие *r* в *T*. В этом случае при вызове [3] клиенту не гарантировалось бы выполнение ожидаемого постусловия β.

Другими словами, *T* как субподрядчик должен выполнять обязательства, взятые исходным подрядчиком *S,* которого только и знают такие клиенты, как *C*.

В этом обсуждении «*a* сильнее *b*» означает (*a* **implies** *b*) **and not** (*a* = *b*). Утверждение «быть слабее» означает обращение приведенной формулы.

Из этих наблюдений следует правило:

Правило переопределения контракта

Переопределяемая версия метода может только: сохранить или ослабить предусловие метода; сохранить или усилить постусловие метода.

Нет необходимости в точном соблюдении контракта. Возможное ослабление предусловия означает, что версия потомка допускает случаи, которые были бы отвергнуты при сохранении исходного предусловия. Усиление постусловия означает, что потомок доставляет улучшенное решение, например, лучшую численную аппроксимацию, в сравнении с исходно обещанным постусловием. Оба случая безвредны, а фактически – могут быть весьма полезными.

Как пример ослабления предусловия приведем предусловие метода *take* класса *TAXI*, которое говорит, что для посадки в такси пассажир должен находиться в радиусе 100 метров от текущей позиции такси. Потомок *DISPATCH_TAXI* ослабляет предусловие этого метода, требуя лишь, чтобы пассажир находился в пределах региона[1]. Понятно, что выполнение этого условия влечет и выполнение исходного предусловия метода из класса *TAXI*.

Как можно в языке программирования задать переопределение контракта? Решение, принятое в Eiffel (и в других нотациях, использующих проектирование по контракту), просто:

- при переопределении метода не разрешается запись предусловия и постусловия в базисной форме – **require** и **ensure**;
- если при переопределении не задавать предусловие и постусловие, то сохраняются условия, заданные родителем. Их можно увидеть в плоском или контрактном облике класса;
- для ослабления предусловия следует использовать предложение в форме **require else** *new_pred*. В этом случае семантика такова: переопределяемый метод имеет предусловие *old_pred* **or else** *new_pred*, где *old_pred* наследуемое предусловие;
- для усиления постусловия следует использовать предложение в форме **ensure then** *new_post*. В этом случае семантика такова: переопределяемый метод имеет предусловие *old_post* **and then** *new_post*, где *old_post* – наследуемое постусловие.

Это решение удовлетворяет правилу, поскольку по правилам логики: «*a implies a* **or** *b*» и «*a* **and** *b implies a*».

[1] Для корректности понимайте под границами региона границы административной области, расширенные на 100 метров.

Семантика **and then**, применяемая для постусловия, является упрощением реально применяемого правила, не имеющего особого значения для нашего обсуждения. Спецификация языка дает полное описание семантики.

Плоский и контрактный облик класса показывают полностью реконструированный контракт, включая наследуемые выражения.

Предусловие *take* в классе *DISPATCH_TAXI*, отражающее наше обсуждение, имеет вид:

```
require else
in_zone: customer.is_in_zone (Current)
```

Много других примеров можно найти при анализе текстов библиотеки EiffelBase.

Контракты в отложенных классах

Правило переопределения контракта дает полную семантику использования контрактов в отложенных классах, что можно видеть на примере *forth*. Отложенные компоненты не имеют реализации, но могут иметь предусловие и постусловие. Отложенный класс, будучи не полностью реализованным, может иметь инвариант класса. Такая стратегия – важная часть того механизма, что делает полезной всю концепцию.

Когда создается отложенный класс и его компоненты, то обеспечивается шаблон, некоторые элементы которого должны быть заполнены потомками (помните про программу с дырами – образец проектирования?). Разрешая потомкам давать свою собственную реализацию, можно, а зачастую *необходимо* наложить ограничения на эту реализацию. Контракты позволяют определить базисную семантику, которую потомки могут уточнить, но не должны противоречить.

Пример для *forth* является типичным:

```
forth
      – Передвинуть курсор к следующей позиции
    require
      in_range: not after
    deferred
    ensure
      increased: index = old index + 1
    end
```

Здесь рассматривается метод, изменяющий положение курсора в списке. *Как* при этом *перемещается* курсор, зависит от реализации. По этой причине метод является отложенным. Но какую бы реализацию не выбрал потомок, он должен работать корректно – для любой позиции, отличной от *after*, индекс курсора должен увеличиваться на 1. Все остальное допустимо до тех пор, пока реализация удовлетворяет этим требованиям.

> По аналогии рассмотрим стереосистему с розетками, куда можно присоединять различные устройства – проигрыватель, плейер, микрофон и другие. Вы можете выбирать устройство, подключаемое к розетке, но только при условии, что оно соответствует указанному напряжению.
>
> Точно так же *search* позволяет подключать различные версии *forth* и других программ, определенных потомками класса *LINEAR*, но при условии, что они удовлетворяют контрактам, заданным в классе *LINEAR* для этих методов.

Это дает нам четкое понимание отложенных классов и их методов. Они не просто избегают ответственности за реализацию, но определяют каркас семантики будущей реализации. Вот почему введение отложенного класса так важно на этапах анализа требований и проектирования – на ранних этапах создания приложения. Отложенные элементы можно использовать для определения базисных свойств – не просто структуры приложения, но и его поведения. Детали будут уточняться в процессе появления потомков, но всегда в полном соответствии с общим каркасом, заданным в начале разработки.

Контракты усмиряют наследование

Как для отложенных, так и для эффективных классов правила адаптации контрактов служат основой корректного использования наследования. Полиморфизм и динамическое связывание являются мощным механизмом, который по этой причине одновременно и опасен. Так как каждый тип может адаптировать наследуемый компонент, как можно гарантировать, что вызов *my_vehicle. turn _ left* после переопределения не заставит ваше транспортное средство поворачивать направо, или останавливаться, или ездить по кругу? Гибкость, которую привносит в программирование комбинация переопределения, полиморфизма и динамического связывания, может зайти слишком далеко. Как проектировщик метода *turn _ left*, требующего левого поворота, вы хотите позволить потомку дать собственную реализацию, но при условии сохранения исходной семантики, гарантирующей левый поворот.

Правило переопределения контракта и связанный с ним механизм языка (**require else …**) обеспечивают нужную степень контроля. Можно указать границы, в рамках которых допустима свобода реализации. Наследование и связанная с ним техника – не просто мощная форма повторного использования, но и техника субподрядов. Классы используют переопределение как субподряд на выполнение некоторых операций потомками. Из-за полиморфизма и динамического связывания клиент не знает, какой субподрядчик будет работать в текущем вызове. Ситуация аналогична покупке iPhone: вы не знаете, где сделана та или иная часть аппарата – в Шанхае, Тайване, Бангалоре или Будапеште. Правило переопределения контракта с успехом может быть названо правилом сохранения честного субподряда.

16.10. Общая структура наследования

Наследование позволяет нам обеспечить общий каркас, где каждый элемент ПО имеет свое четкое место. Большинство ОО-языков (значимым исключением является C++) определяют специальный класс – общего предка всех классов, иногда называемого Object (Smalltalk, Java, C#). В Eiffel такой класс называется *ANY*. Он входит в библиотеку «Kernel», которая содержит также некоторые фундаментальные классы, тесно связанные с определением языка (*ARRAY, STRING*, базисные типы данных – *BOOLEAN, INTEGER, REAL, CHARACTER*).

Показанные на следующем рисунке классы *A, B* могут быть классами, написанными вами или мной. Правило, определяющее роль *ANY*, просто: любой класс, в котором даже не задано предложение наследования **inherit**, записанный в виде

```
class A feature … end
```

понимается, как если бы он был явно записан в форме

```
Class A inherit ANY feature … end
```

Общая структура наследования показана на рисунке:

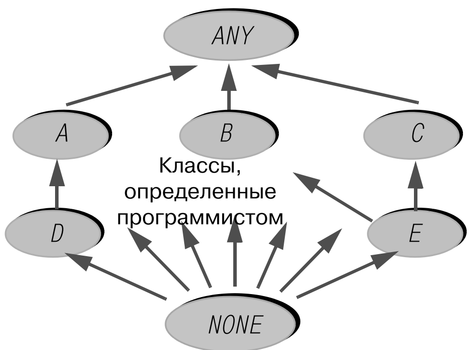


Рис. 16.15. Общая структура наследования

Имеет место следующее свойство:

Теорема об универсальном наследовании и согласовании (Eiffel)

Каждый класс является наследником *ANY.*
Каждый тип согласован с *ANY*.

Класс *ANY* включает общецелевые компоненты, полезные для всех классов: *is_equal* и другие полезные методы. Мы уже встречались с использованием метода *print* этого класса, который позволяет печатать представление, заданное по умолчанию для любого объекта. Тип *ANY* является наиболее общим типом, с ним согласован любой другой тип. Переменная, объявленная как *v*: *ANY*, является полиморфной и может обозначать объекты любого типа.

В нижней части рисунка показан еще один предопределенный класс *NONE*. Если *ANY* является прародителем всех классов, то *NONE* является потомком всех классов. В отличие от полнокровного класса *ANY*, обладающего многими полезными компонентами, класс *NONE* – это удобная фикция, закрывающая снизу структуру наследования, но без компонентов, имеющих смысл. Он служит двум полезным целям.

- Как тип, он позволяет задавать тип **Void** – предопределенное значение, представляющее ссылку void (значение, не связанное ни с каким объектом).
- Как класс, он поддерживает скрытие информации. Как отмечалось ранее, мы объявляем скрытые компоненты класса, используя предложение в форме **feature** { *NONE* }. Формально это означает, что компонент экспортируется только классу *NONE* – фактически, никакому классу.

Данный синтаксис задает специальную форму выборочного экспорта. Предложение **feature** можно задавать в форме **feature** {*C, D, ...*}, означающей, что компоненты, объявленные в разделе **feature**, экспортируются классам, указанным в скобках, и их потомкам.

16.11. Множественное наследование

В самом начале обсуждения наследования отмечалось, что класс может иметь двух и более родителей – случай, известный как множественное наследование. Это важное свойство, ко-

торое может давать серьезные преимущества. Следует сделать предостережение в связи с утверждениями, с которыми можно встретиться при чтении наивной литературы по ОО-программированию.

Почувствуй методологию

Развенчание легенды о сложности множественного наследования
Множественное наследование относится к базисным механизмам, применяемым для конструирования надежного, масштабируемого, повторно используемого ПО (конечно, как и для других мощных механизмов, возможно его неверное применение). Не следует поддаваться заблуждениям, что эта техника трудна или проблематична.

(Если вы новичок в этой области, то это предостережение к вам не относится, – тот, кто не ведает, тот и не заблуждается)

Неверное понимание сложности множественного наследования идет от ранних ОО-подходов, когда страдала техника реализации множественного наследования. Теперь все это в прошлом, и сразу же, как только обнаруживается практическая польза множественного наследования, становится трудно представить, как можно жить без него.

Применение множественного наследования

Наследование является ***специализацией***: транспорт – специальный вид движущегося объекта, такси – специальный вид транспорта. Иногда некоторое понятие является специализацией двух или трех понятий, тогда необходимо множественное наследование. Без него пришлось бы выбрать из понятий только одного родителя, дублируя код для остальных.

Чтобы установить наследование от нескольких классов, достаточно просто перечислить их в части **inherit**, задав для каждого класса при необходимости переопределение компонентов и другие предложения адаптации наследования, если таковые имеются:

```
class TROLLEY inherit
    TRAM
      redefine add_station, remove_station end
    BUS
feature
…
end
```

Простой пример может быть найден в базисных библиотеках. Класс *NUMERIC* задает объекты, поставляемые со стандартными математическими операциями:

```
deferred class NUMERIC feature
    plus alias "+" (other: NUMERIC): NUMERIC deferred end
    minus alias "-" (other: NUMERIC): NUMERIC deferred end
    times alias "_" (other: NUMERIC): NUMERIC deferred end
    divided alias "/" (other: NUMERIC): NUMERIC deferred end
    …
end
```

(Это только набросок. Полный текст класса можно увидеть в EiffelStudio)

Еще один библиотечный класс, *COMPARABLE*, задает объекты, которые поставляются с операциями отношениями, задающими полный порядок:

```
deferred class COMPARABLE feature
    lesser alias "<" (other: NUMERIC): BOOLEAN deferred end
    lesser_or_equal alias "<=" (other: NUMERIC): BOOLEAN do …end
    greater alias ">" (other: NUMERIC): BOOLEAN do …end
    greater_or_equal alias ">=" (other: NUMERIC): BOOLEAN do …end
    …
end
```

Не все типы, допускающие сравнение, являются численными. На строках определено отношение полного порядка, так называемый лексикографический порядок, но не определены операции умножения и деления. Не все числовые типы являются сравнимыми – нет общепринятого отношения полного порядка для комплексных чисел или матриц. Но некоторые типы, такие как *INTEGER* или *REAL*, обладают и теми и другими свойствами. Соответствующие классы используют множественное наследование, отражая этот факт:

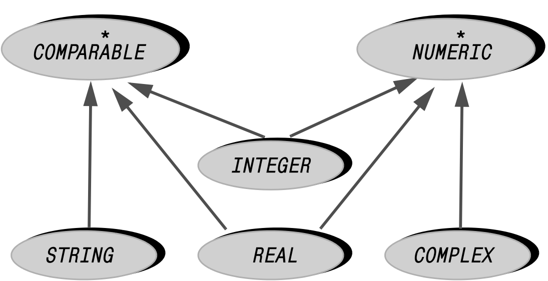


Рис. 16.16. Множественное наследование

Такие языки, как Java и C#, допускают множественное наследование, но не классов, а интерфейсов, подобных, как отмечалось ранее, полностью отложенным классам. Следующий пример иллюстрирует разницу. Класс *COMPARABLE* нуждается только в одном отложенном свойстве, например *lesser* (alias <). Все остальные могут быть определены как эффективные компоненты, например:

```
lesser_or_equal alias "<=" (other: NUMERIC): BOOLEAN
      — Является ли текущий объект меньше или равным other?
    do
      Result := (Current < other) or (Current ~ other)
    ensure
      definition: Result = ((Current < other) or (Current ~ other))
end
```

Аналогично *greater*(*other*) определяется как *other*.*lesser*(**Current**). Нетрудно определить и отношение «больше или равно». Класс не просто задает начальную реализацию, но и устанавливает отношения, существующие между операциями, что отражается в постусловиях.

Класс *COMPARABLE* является примером образца программы с дырами – схемы, которая оставляет некоторые операции для реализации потомкам; здесь в качестве таковой используется операция *lesser* – хотя эту роль могла играть любая из четырех операций – и определяет остальные операции в терминах отложенных операций.

Если же приходится выбирать между интерфейсом, полностью отложенным, и классом, полностью эффективным, то теряется мощный инструмент проектирования. Более точно, сделав *COMPARABLE* интерфейсом, вы заставляете каждого потомка дать реализацию всех методов. Это может приводить:

- к дублированию кода, так как все реализации *lesser_or_equal* будут идентичны, если используют приведенный выше код;
- к увеличению риска появления ошибки, так как у авторов исходного класса помимо контрактов нет способов воздействия на реализацию, даваемую потомками.

Разделение на классы и интерфейсы является искусственным, ограничение множественного наследования интерфейсами является непрактичным.

Переименование компонентов

Множественное наследование приводит к проблемам перегрузки, если класс С наследует от двух классов с идентично названными именами компонентов:

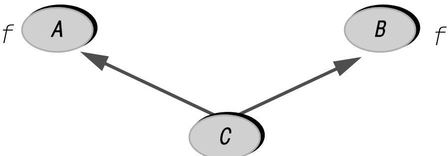


Рис. 16.17. Конфликт имен

Конфликт легко разрешается, если мы по-прежнему избегаем перегруженных методов, когда одно и то же имя дается нескольким методам класса. Попытаемся скомпилировать код, соответствующий приведенной выше структуре:

```
class C inherit
    A
    B
end
```

Здесь *A* и *B* имеют компоненты, названные *f*. Компиляция класса *C* в этом случае не пройдет. Необходимо *переименование*:

```
class C inherit
    A rename f as first_f end
    B
end
```

Предложение **rename** (которое может быть скомбинировано с переопределением: **rename** *f* **as** *first_f* **redifine** *first_f* **end**) просто указывает, что компонент, известный в *A* как *f*, будет известен в классе *C* под именем *first_f*. Конечно, мы могли бы переименовать компонент и в классе *B* или в обоих классах.

Переименованный компонент все же остается прежним компонентом – компонентом, ранее известным как *f* в *C*. Так что для *a1: A; c1:C* следующие вызовы оба являются правильными:

```
a1.f
c1.first_ f
```

Конечно же, возникнет ошибка при вызове *a1. first_f*, поскольку понятно, что класс *A* не знает имя *first_f*. Вызов *c1.f* синтаксически правилен, но ссылается на компонент из класса *B*. Если бы и этот компонент был переименован, то и этот вызов был бы ошибочным. В полиморфной ситуации после присваивания *a1:= c1* два приведенных выше вызова имели бы одинаковый эффект, так как *a1* и *c1* обозначали бы один и тот же объект, а *f* и *first_f* обозначали бы один и тот же компонент в соответствующих классах.

Плоский и контрактный облик класса отражают эффект как переименования, так и переопределения.

Помимо устранения конфликта имен, переименование позволяет дать имена, согласованные с общим принципом именования, который принят в классе. Иногда имена, принятые у родителя, не отвечают контексту наследника: тогда, благодаря переименованию, их можно адаптировать, сделав их более благозвучными для слуха наследника.

Переименование в сравнении с переобъявлением

> Переименование сохраняет наследуемый компонент, изменяя его имя. Переобъявление, сохраняя имя, изменяет компонент (через переопределение, эффективизацию или отмену определения).

Когда требуется изменить и сам компонент, и его имя, допускается комбинация переименования и переобъявления.

Эти приемы являются центральными в использовании наследования как инструмента архитектора ПО. Как отмечалось ранее, проектирование класса родственно проектированию машины. Наследование – это инструмент, помогающий создавать машины путем расширения возможностей и специализации уже существующих машин. Эта техника конструирования дает больше, чем техника, построенная на интерфейсах, поскольку клиентам класса нет необходимости знать (за исключением полиморфного его использования), как был получен класс – путем наследования или построен с нуля.

От множественного наследования к повторному наследованию

Необходимо проанализировать один технический случай – повторное наследование, возникающее в результате множественного наследования, когда от потомка идет несколько путей к предку:

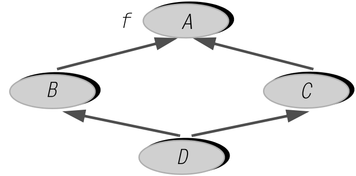


Рис. 16.18. Повторное наследование

Такие структуры возникают при продвинутой разработке. Примеры можно найти в библиотеке EiffelBase. Правило следует знать, поскольку повторное наследование всегда возникает при множественном наследовании, хотя бы за счет того, что у всех классов существует общий предок – *ANY*.

В связи с повторным наследованием возникают два вопроса: судьба повторно наследуемых компонентов и возможная неоднозначность при динамическом связывании.

Для случая, изображенного на рисунке, первый вопрос, который следует задать относительно *f* – компонента из класса *A*: в классе *D* ему должны соответствовать два компонента или один? Ответ прост и соответствует духу предыдущего обсуждения.

- Если компонент приходит с обеих сторон под тем же самым именем, то есть он не переименовывался на всем пути наследования, оставаясь известным как компонент *f*, то ясно, что речь идет об одном и том же компоненте. Это случай допустимого конфликта имен: хотя родители имеют компоненты с одним и тем же именем, проблемы не возникает, поскольку речь идет об одном и том же компоненте. Этот случай известен как **разделение компонента** или склеивание.
- Если же компонент наследуется под двумя различными именами – как результат переименования на пути наследования, – то снова, во избежание любой перегрузки, он должен обозначать два различных компонента. Мы говорим в этом случае о **репликации** компонента.

Очевидное ограничение применимо к случаю разделения. Если на одном из путей встретилось переобъявление, то версия компонента на другом пути должна быть отложенной (исходно или в результате отказа от определения). Если же на обоих путях мы сталкиваемся с двумя эффективными методами и хотим сохранить обе версии, то одну из них нужно переименовать, возвращаясь назад к случаю репликации.

При репликации может возникнуть неоднозначность, когда встречается переопределение. Такая ситуация может встретиться, например, при повторном наследовании от *ANY*, если один из классов вдоль пути наследования вводит свое собственное понятие копирования и эквивалентности, переопределив *copy* и *is_equal* из класса *ANY*:

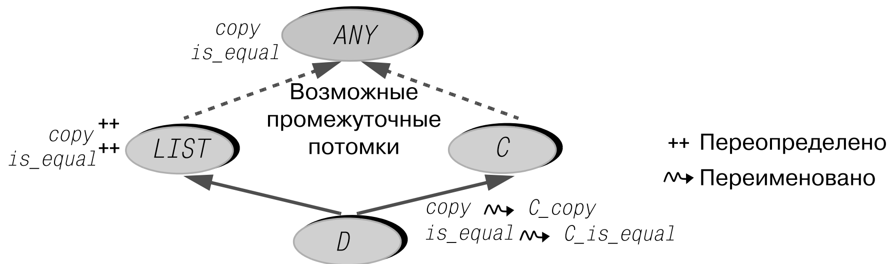


Рис. 16.19. Необходимость «select»

Методы *copy* и *is_equal* должны всегда переопределяться совместно, поскольку постусловие *copy*(*other*) устанавливает *is_equal*(*other*). Любая операция копирования должна убеждаться, что результат эквивалентен цели копирования в соответствии с локальным определением эквивалентности.

Класс *LIST* переопределил *copy* и *is_equal,* чтобы быть уверенным, что копируется не заголовок списка, а его содержимое (большинство контейнерных классов подобным же образом определяют понятие копирования и сравнения).

Теперь *D* наследует эти методы от *LIST,* а также от класса *C*, который сохранил версии по умолчанию, наследованные от *ANY*. Все это работает в соответствии с предыдущими правилами. Класс *D* должен переименовать компоненты во избежание конфликта имен, чтобы они дублировались.

Проблема возникает в связи с **полиморфизмом.** Рассмотрим *a* типа *ANY* и *d1* типа *D*. Выполним присваивание, а потом вызов:

```
a := d1
a.copy (…)
```

В этом случае при динамическом связывании возникает неоднозначность: для родительского метода *copy* у потомка имеются две реализации. Возникает вопрос: следует ли для связывания использовать версию *LIST* (известную как copy в классе) или *C*-версию (*C_copy*)? Ситуация возникает всякий раз, когда одновременно встречается репликация и переопределение. Для устранения неоднозначности требуется ввести предложение **select**. Вот как следует записать класс *D*:

```
class D inherit
      LIST [T] select copy, is_equal end
      C rename copy as C_copy, is_equal as C_is_equal end
feature
      … Остаток текста класса …
end
```

Эта запись говорит, что для полиморфной цели с возможной неоднозначностью при динамическом связывании следует выбирать версию из *LIST*. Без предложения **select** в таких ситуациях не обойтись.

Конечно, вы могли бы остановить свой выбор на другой ветви либо некоторые компоненты выбирать от одного родителя, другие – от другого.

16.12. Универсальность плюс наследование

Введение наследования позволяет нам повторно обратиться к рассмотрению другого главного механизма расширяемости классов – универсальности. Независимо оба механизма уже рассмотрены, их комбинация добавляет новый потенциал.

Полиморфные структуры данных

У нас уже появлялся пример сотрудничества этих механизмов – полиморфные структуры данных. Рассмотрим контейнер, такой как:

```
fleet: LIST [VEHICLE]
```

Элементы в контейнере могут быть экземплярами любого из потомков *VEHICLE*:

На следующем рисунке показан графический образ комбинации «универсальность-наследование». Хотя никаких новых концепций не вводится, но здесь иллюстрируется неформальная интерпретация взаимодействия двух механизмов.

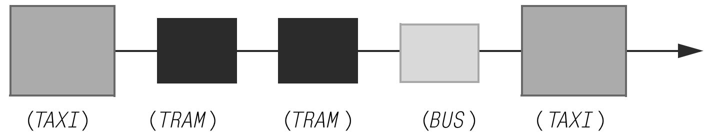


Рис. 16.20. Полиморфный список

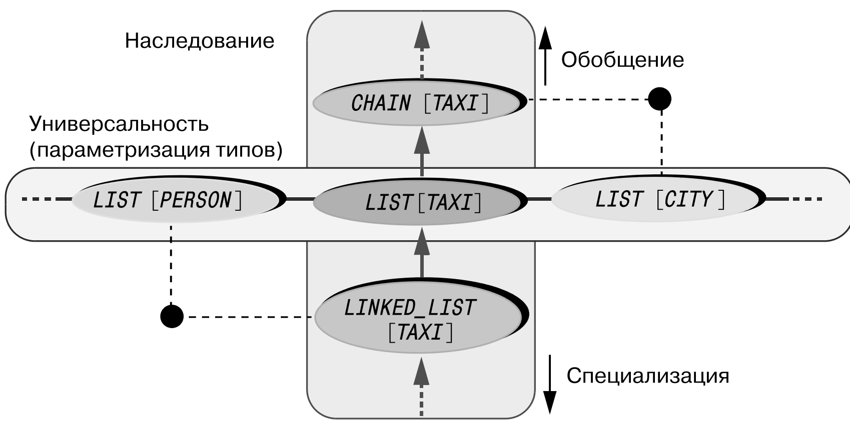


Рис. 16.21. Наследование и универсальность

Начнем с центра, где показан класс, задающий понятие «список такси», – это класс с ожидаемыми компонентами, характерными для списка: добавить и удалить элемент, передвинуть курсор, получить элемент и так далее. Мы можем перейти теперь к концепции нового класса, двигаясь вдоль двух различных направлений.

- Помимо списка *такси* интерес могут представлять списки *персон*, *городов*, *объектов* любого другого типа. **Универсальность** позволяет нам путешествовать по горизонтали, будучи поддержана механизмом параметризации. Важно то, что удается избежать дублирования кода и гарантировать безопасность типов, о чем говорилось в исходном обсуждении этого механизма.
- *Список* является специальным случаем более общего понятия «*цепочка*». В свою очередь, специализированным вариантом списка является «*связный список*», обладающий функциональностью списка, но учитывающий особенности реализации. **Наследование** позволяет нам путешествовать по вертикали. Оно поддерживается механизмами обобщения и специализации, представляя опять-таки мощную форму повторного использования.

Рисунок иллюстрирует возможность свободного перемещения по метафорической плоскости, переходя от «списка персон» к «цепочке городов».

Ограниченная универсальность

Полиморфные структуры данных не являются единственным способом комбинирования универсальности и наследования. Другая важная техника следует из исследования вопроса: *что можно делать с сущностями или выражениями универсального типа?*

В универсальном классе, таком как *LIST*[*G*] или *ARRAY*[*G*], в тексте этого класса, как правило, появляется объявление *x*: *G*. Какие операции применимы к *G*?

Прежнее обсуждение может подсказать ответ. Так как *G* – просто держатель места для любого типа, который будет задан фактическим родовым параметром (*VEHICLE*, например), применимыми операциями могут быть только операции доступные любому типу, следовательно – операции, введенные в классе *ANY*. Так что можно использовать вызовы: *x.cloned, x.is_equal*(*y*) и так далее, но нельзя использовать компоненты, отсутствующие у *ANY*.

Что если требуется больше операций? Рассмотрим случай «сортирующего» класса с методом *sort*, сортирующим элементы структуры данных, – массива, списка, например, списка целых. Естественно, хочется иметь механизм, работающий для многих *типов*, а не только для целых. Универсальность кажется подходящим механизмом для реализации поставленной задачи.

Алгоритмы сортировки – это обширная область информатики, она не является предметом нашего изучения, хотя топологическая сортировка была рассмотрена достаточно подробно. Но нет необходимости в разборе различных методов сортировки для осознания того факта, что для сортировки, предположим, массива, понадобится выполнять следующие операторы:

```
x := t[i] ; y := t[j]
if x < y then                                                          [4]
            – Обмен элементов, находящихся в позициях i и j
    a [i] := y ; a [j] := x
end
```

Алгоритм будет выполнять обмен значениями для элементов, нарушающих порядок. Не интересуясь сейчас тем, как найти такие элементы, сосредоточимся только на одном вопросе: а как выполнить само сравнение, какую операцию «<» следует использовать?

Если речь идет о сортировке целых, то все понятно, но что если необходимо ранжировать игроков в теннис, – как тогда выполняется сравнение? Как быть, если в общем случае мы даже не знаем, есть ли вообще такая операция у объектов?

Иногда мы можем это знать, иногда нет. Как отмечалось при рассмотрении класса *COMPARABLE*, общепринятого тотального порядка не существует на множестве комплексных чисел или матриц.

Чтобы обеспечить общецелевой алгоритм сортировки, применимый для сортировки массивов с элементами многих типов, можно было бы поместить приведенный выше код **[4]** в универсальный класс

```
class SORTER [G …] feature
   sort_array (a: ARRAY [G])
            – Сортировка элементов a в соответствии с отношением порядка.
      local
            x, y: G
      do
            … Код, такой, как [4] с проверками, такими, как x < y …
      end
   …
end
```

Но как быть с операцией «<»? Ведь *x* и *y* объявлены как сущности типа *G*, так что над ними определены только операции из *ANY*, а там нет операции, позволяющей сравнивать объекты. У нас есть потребность в использовании операции сравнения, но она доступна только для специфических классов, таких как класс *COMPARABLE*.

Классы, такие как класс *COMPARABLE?* Почему бы не выбрать сам *COMPARABLE*? Он отложен, и его эффективные наследники обеспечат реализацию *lesser* **alias** "<". Можно пойти дальше и полагать, что любой класс, объекты которого удовлетворяют отношению тотального порядка, должен быть потомком *COMPARABLE*. Тогда ответ на наш вопрос становится очевидным: формальный родовой параметр *G* не должен быть более произвольным типом, он должен задавать тип, согласованный с *COMPARABLE*. Следующий синтаксис позволяет выразить это свойство:

```
class SORTER [ G -> COMPARABLE] feature
            … Остаток как выше …
```

Символ -> соответствует стрелке на диаграмме наследования, указывая на родительский класс (здесь *COMPARABLE*). Вся конструкция в целом задает **ограничение универсальности**. Смысл ограничения в том, что теперь родовое порождение *SORTER*[*T*] является правильным только при условии, что Т удовлетворяет ограничению. Так что *SORTER*[*INTEGER*] и *SORTER*[*STRING*] прекрасно подходят, так же как и *SORTER*[*TENNIS_PLAYER*], при условии, что *TENNIS_PLAYER* наследует от *COMPARABLE*. Но не подходят *SORTER* [*COMPLEX*] и *SORTER* [*VEHICLE*], если мы только не сделаем транспортные средства сравнимыми. Неподходящие случаи будут отвергнуты на этапе компиляции.

Как вы догадываетесь, базисный случай *LIST* [*G*] (неограниченная универсальность) можно рассматривать как краткую форму записи *LIST* [*G-> ANY*].

Ограничение универсальности имеет много применений. Некоторые часто встречающиеся случаи используют отложенные библиотечные классы, подобные *COMPARABLE*.

- При определении классов, задающих вектора и матрицы, разумно поставлять их с компонентами, реализующими сложение и другие числовые операции. Например, должно быть возможно вычислять *m1* + *m2*, где *m1* и *m2* относятся к типу *MATRIX*[*T*]. Учитывая тип *NUMERIC*, к решению приводит объявление: *MATRIX*[*T->NUMERIC*]. В этом случае можно использовать *MATRIX*[*INTEGER*], но не *MATRIX*[*STRING*]. Стоит ли сделать сам класс *MATRIX* наследником *NUMERIC*? Этот интересный прием имеет смысл, так как обеспечивает все требуемые операции (модель *NUMERIC* соответствует математическому понятию *кольца*). В этом случае становятся возможными такие родовые порождения классов, как *MATRIX*[*MATRIX*[*INTEGER*]] или *MATRIX*[*MATRIX*[*MATRIX*[*REAL*]]] и так далее.
- Часто для универсального класса *C* [*T*] необходима возможность сохранять элементы типа *T* в хеш-таблице. Это предполагает, что для каждого такого элемента можно вычислить целочисленную хеш-функцию. Требование простое, но не все типы ему удовлетворяют, нужно иметь дело в этом случае с потомками класса *HASHABLE*, которые задают реализацию отложенного запроса *hash_code*.

Сам класс *HASH_TABLE* объявляется как

```
class HASH_TABLE [ELEMENT, KEY -> HASHABLE] feature …
```

Теперь становится понятным, почему родовой параметр в классе *TOPOLOGICAL_SORTER* также был ограничен *HASHABLE* – мы хотели помещать элементы в хеш-таблицу.

Объявление *HASH_TABLE* демонстрирует, что универсальный класс может иметь несколько параметров, некоторые из которых могут быть ограниченными. Также ясно, что имена формальных параметров можно выбирать по собственному вкусу.

Замечание: можно задавать множественные ограничения универсальности. Например, можно объявить класс:

```
class C [G -> {COMPARABLE, NUMERIC, HASHABLE}] feature
```

Это ограничение говорит, что фактический тип при родовом порождении должен соответствовать всем перечисленным типам. Правильным примером может служить класс *INTEGER*. Пример немного экстремальный, но требование сравнимости элементов и возможности выполнять над ними числовые операции встречается достаточно часто.

Ограниченная универсальность иллюстрирует фундаментальную роль типов в современном программировании. Для решения обсуждаемых проблем возможны и другие подходы, такие как передача процедуры, выполняющей сравнение, методам класса *SORTER*. Но данный подход наиболее согласуется с ОО-идеями – сделать требуемую функциональность частью типа. Это также означает, что по-прежнему можно полагаться на компилятор, который будет выполнять все необходимые проверки корректности, что позволит избежать появления подобных ошибок в период выполнения. Если компилятор отвергает ваш класс из-за несогласованности типов, помните, что эта новость, кажущаяся *плохой,* по-настоящему является *хорошей*, – лучше вы поймаете «жучка», чем он поймает вас.

Цените мощь статической типизации. Это настоящий механизм верификации программ. Изощренность системы типов, которой вы теперь владеете, – с классами, ограниченной и неограниченной универсальностью, включающей множественные ограничения, одиночное, множественное и повторное наследование, полиморфные переменные и структуры данных, правила вызова методов, передачи аргументов и присваивания, – определяет каркас с математической строгостью, который компилятор использует для выполнения критически важных согласованных проверок. Типы элементов вашей программы отражают семантику, лежащую в основе строящейся модели внешнего мира.

16.13. Обнаружение фактического типа

Помните вопрос, задаваемый в начале этой главы: «Что, если я знаю, что последний элемент списка является экземпляром *TAXI*, а не просто *VEHICLE,* и хочу применить к нему операцию, специфическую для такси»? Обсуждение вклада динамического связывания в архитектуру ПО объясняет, почему для полиморфных структур данных нет срочной необходимости в его рассмотрении.

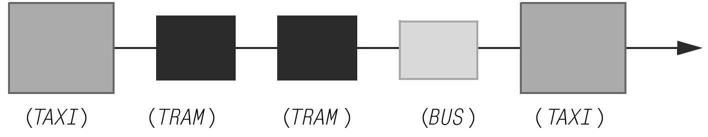


Рис. 16.22. Полиморфный список

Вы рассматриваете элемент списка, известного вам как список транспортных средств:

```
fleet: LIST [VEHICLE]
```

Такое объявление отвергает специфику – идентичность, характерную для такси, автобуса или трамвая, – перед нами просто транспортные средства. У такого подхода есть достоинства и недостатки. Недостаток в том, что нельзя использовать преимущества знания «трамвайности» или «таксичности». Достоинство подхода в том, что можно применить операцию, допустимую для транспортных средств, без знания того, какое именно средство используется в данном случае, даже если эта операция выполняется специфическим образом для каждого вида транспортного средства. Так что всякий раз, когда мы хотим применить специфическую операцию, обычно необходимо сделать эту операцию применимой для всех транспортных средств и задать специфическую реализацию для каждого вида. Это единственный способ справиться с синдромом множества явных вариантов.

Такой подход неприменим в двух случаях.

- Операция, которую мы хотим использовать, настолько специфична для выбранного типа, что ее никак нельзя применить на более высоком уровне иерархии.
- Ваша программа получает объекты из внешнего мира, из источника, не подлежащего вашему контролю, – из файла, базы данных, сети. Единственное, что известно, – что объекты могут быть самого общего типа (для нас это тип *ANY*).

Рассмотрим важный пример второго случая. Любое хорошее ОО-окружение поддерживает механизм сериализации – записи структуры объектов в файл, и десериализации – получения их оттуда.

```
retrieved: ANY
            -- Объект, полученный последней операцией десериализации
```

Объект можно объявить только типа *ANY*, поскольку операция десериализации общецелевая. Она должна работать для любой проблемной области и возвращать любой извлекаемый из файла объект, будь то такси, трамвай, город или еще что-нибудь. В конкретном приложении и для конкретного файла можно ожидать определенный тип объекта, например, такси, но нет возможности, задав *t*: *TAXI*, просто написать:

```
t := my_serializer.retrieved
```

Тип *ANY* не согласован с *TAXI* – справедливо лишь обратное утверждение. В этой ситуации необходимо явное преобразование, называемое кастингом или приведением типа. Зная, что полученный объект является такси, мы должны вернуть ему индивидуальность, отвергнув идентификацию его как *ANY*, присущую всем объектам в этой толпе. Для поддержки процесса идентификации нам необходимо нечто большее, чем наследование, полиморфизм и динамическое связывание.

В поисках нового механизма заметим, что кастинг не может быть безусловным. Когда объект приходит из файла или по сети, можно ожидать, что он принадлежит определенному типу, но быть уверенным в этом нельзя. Теперь мы имеем дело с объектами, не подлежащими полному контролю в программе. Когда объекты создаются программой, объявление *t*: *TAXI* гарантирует, что *t* всегда будет присоединен к экземпляру *TAXI*. Для объектов, приходящих из дикого внешнего мира, таких гарантий нет.

Даже, если структура объектов была сериализована вами, при ее десериализации нет гарантии, что она сохранила девственность, – файл мог быть поврежден случайно или злонамеренно. Отсюда для релевантного механизма языка следует правило:

Почувствуй методологию

Принцип Кастинга
Любой механизм приведения типа объекта, заданного ссылкой, приходящего без соответствующих статических правил согласования, должен основываться на проверке периода выполнения, чтобы убедиться, что тип объекта согласован с ожидаемым.

Другими словами, такой механизм должен быть условным. Если вы ожидаете *TAXI*, а обнаружили некий другой тип, не согласованный с такси, попытка кастинга должна закончиться неуспехом.

Механизм, не удовлетворяющий этому принципу, свойственен кастингу, реализованному в языке C (в языке C++ принцип также не выдерживается, но там есть более изощренная форма приведения к типу, соответствующая принципу). В языке C, написав (T) e, где T – тип, а e – ссылка, получим ссылку на объект типа T с сохранением значения e, независимо от фактического типа данных, хранящихся в соответствующей памяти. Это прямое следствие машинного уровня языка, который рассматривает ссылку (указатель) просто как адрес памяти, а не как ссылку на объект определенного типа. Логика такова: «программисты знают, что они делают».

Общим термином для описания механизма кастинга, удовлетворяющего принципу, является «**динамический** кастинг» (приемлем также термин «условный кастинг»). Это еще одна область, где отсутствует стандартная терминология. Можно встретить в литературе «сужение типа» (narrowing) или «приведение вниз» (downcasting). Эти термины используются, когда речь идет о приведении общего типа к специальному типу, например, от *VEHICLE* к *TAXI*. Такой случай часто встречается на практике, но он не единственный. Мы увидим, почему изучаемый механизм может применять динамический кастинг к объектам произвольных типов.

Наиболее общий термин, покрывающий все случаи нахождения типа объектов во время выполнения, носит название **RTTI** (Run Time Type Identification).

Прошу прощения за громкие термины, не мной они введены. Вы должны знать, что они существуют, но что действительно следует знать, – это лежащие в основе концепции и общее решение, которое сейчас будет представлено.

Давайте рассмотрим два механизма динамического кастинга, один – текущий, другой – устаревший, но все еще используемый.

Тест объекта

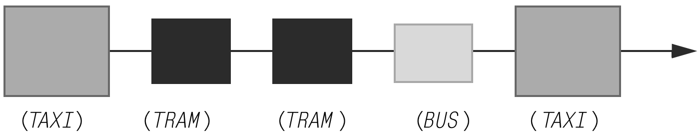


Рис. 16.23. Полиморфный список

Предположим, вы убеждены, что последний объект *fleet* – это объект *TAXI*, как на рисунке. Потому вы хотите применить специфический компонент, такой, как *take*, к этому элементу. Простейшее решение не будет работать:

```
fleet.last.take (…)
```

Причина этого должна быть ясна, но ее можно проанализировать тщательнее, если разделить выполнение оператора на части:

```
– Предыдущие объявления:
fleet: LIST [VEHICLE]
t: ?                    – Держатель места, должен быть заменен фактическим типом
…
t := fleet.last                                                              [5]
t.take (…)                                                                   [6]
```

При объявлении *t* использован знак вопроса вместо фактического типа. Причина в том, что возникает неразрешимая дилемма.

- Если объявить *t* как *VEHICLE,* то **[5]** будет правильной конструкцией, а **[6]** нет, так как take не является компонентом *VEHICLE.*
- Если объявить *t* как *TAXI,* то **[6]** будет правильной конструкцией, но не **[5]**, так как нарушается правило полиморфизма типов: потомку нельзя присваивать родительский объект – это противоречит направлению согласования типов.

Конструкция **теста объекта** обеспечивает решение в таких случаях. Тест объекта – это булевское выражение, которое (как форма RTTI) определяет, согласован ли тип объекта с ожидаемым типом. В случае согласования возникает дополнительный эффект – появление локальной переменной, присоединенной к объекту. Применяя тест объекта в нашем примере, получим:

```
if attached {TAXI} fleet.last as t then
    t.take (…)
    … Любая другая TAXI-операция над t, присоединенному к TAXI-объекту …
else
    …Делай что-нибудь еще (не над такси), если необходимо …
end
```

В соответствии с принципом кастинга операция является условной. Она тестирует тип *fleet.last* – фактического объекта, присоединенного к ссылке во время выполнения.

Возвращается значение **false**, если нет согласования с заданным типом *TAXI.* В этом случае будет выполняться **else**-ветвь данного оператора, если она присутствует. Если же тип согласован, то в нашем распоряжении появляется объект такси, и он становится локально доступным через заданное имя *t*, так называемую локальную переменную теста объекта. По семантике это соответствует корректному объявлению *t* и присвоению ему значения, как в **[5]**.

Если тест объекта служит условием оператора **if**, как в данном случае, то *областью* локальной переменной теста является **then**-ветвь, где можно использовать *t* как переменную *TAXI*, обозначающую значение *fleet.last.* Поскольку это и в самом деле *TAXI*, что установлено как статически в результате объявления, так и динамически – в результате проверки присоеди-

ненного объекта во время выполнения, то без всякого риска можно применять к *t* операции, специфические для такси.

Рассмотрим, как определяется область локальной переменной теста, где безопасно можно выполнять операции. Следующие случаи покрывают все практические потребности (если возникает более сложная ситуация, можно ввести свою локальную переменную).

- Как мы видели, если тест объекта появляется как условие **if**, то областью является **then**-ветвь оператора. Сюда же включается случай, когда тест комбинируется с другими условиями через **and then**, как **if attached**{ *TAXI* }*fleet.item* **as** *t* **and then** *v.is_moving* **then** ...
- Если используется отрицание теста (возможно, комбинируемое с другими условиями через связку **or else**), то *областью* является **else**-ветвь, как **if not attached**{ *TAXI* }*fleet.item* **as** *t* **then** ...
- Аналогично, если тест появляется с отрицанием в условии выхода из цикла – предложении **until**, то *областью* является тело цикла. И снова возможна комбинация теста с другими булевскими выражениями.

Как пример последнего случая, рассмотрим, как подсчитать число объектов, предшествующих первому объекту специального типа в полиморфном списке.

```
pre_taxi_count (fleet: LIST [VEHICLE]): INTEGER
      — Число объектов fleet, перед первым TAXI.
    do
      from fleet.start until
                     fleet.after or else attached {TAXI} fleet.item as t
      loop
                              Result := Result +1 ; fleet.forth
      end
    ensure
      non_negative: Result >= 0
      at_most_length_of_list: Result <= fleet.count
    end
```

Этот конкретный алгоритм не использует локальную переменную теста *t*; как следствие, тест можно записывать в этом случае проще, как **attached**{*TAXI*} *fleet.item*.

Ограничений на типы, включаемые в тест, не существует: **attached**{*U*} *exp* **as** *x*, где *exp* является выражением (статического) типа *T* – типы *U* и *T* могут быть произвольными. В наиболее распространенном случае, названном «приведением вниз», тип *U* является потомком типа *T*. Пример с типами *VEHICLE* и *TAXI* иллюстрирует эту ситуацию. Но это не является абсолютным требованием, и для множественного наследования может понадобиться тест для типов *U* и *T*, не связанных отношением наследования. Типичный пример показан на следующем рисунке.

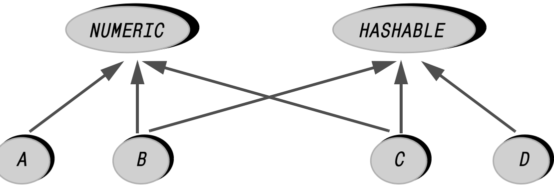


Рис. 16.24. Непрямые отношения наследования

Классы *NUMERIC* и *HASHABLE* разделяют потомков, таких как *B* и *C*. Если имеется список *numlist* из элементов *NUMERIC*, то может потребоваться хеширование тех элементов списка, которые допускают эту операцию:

```
if attached {HASHABLE} numlist.item as h then
    your_hash_table.put (h, h.hash_code)
end
```

В этом фрагменте *numlist* имеет тип *LIST*[*NUMERIC*], а *hash_code* – это метод класса *HASHABLE*. Приведение типов в данном случае не является приведением вниз, так как *HASHABLE не согласован с NUMERIC*. Случай вполне законный и реально существующий. Он иллюстрирует, почему необходим общий механизм динамического кастинга, а не ограниченное сужение типа.

Попытка присваивания

(Этот раздел является дополнительным и может быть пропущенным при первом чтении, но уже следующий раздел «Разумно используйте динамический кастинг» прочесть необходимо)

Тест объекта вытесняет старый механизм – попытку присваивания. Дадим короткий обзор этого механизма. Вернемся к примеру, показывающему неспособность базисного присваивания обеспечить приведение вниз.

```
fleet: LIST [VEHICLE]
t: TAXI
…
– t := fleet.last             – Только для сравнения: закомментировано, поскольку
                                                              – неверно!
                                                                               [7]
t?= fleet.last
t.take (…)                    – Правильно, но не безопасно, смотри далее
```

Мы объявили *t* как *TAXI*, что делает **[7]** неверным: присваивание в направлении, ошибочном для согласования типов. Вместо обычного присваивания (:=) давайте будем использовать «попытку присваивания», использующую символы «?=». Семантика такова:

- если во время выполнения значение источника (правая сторона присваивания) согласовано по типу с типом цели (здесь *TAXI*), то цель, здесь *t*, присоединяется к объекту как в обычном присваивании;
- в противном случае *t* получает значение void.

Безопасное использование попытки присваивания должно непосредственно после попытки присваивания и до использования полученной переменной как цели вызова компонента выполнить «тест на void»:

```
t ?= fleet.last
if t /= Void then
    t.take (…)                – Правильно и безопасно
    … Любая другая TAXI операция над t, присоединенному к TAXI-объекту…
else
    … Делай что-нибудь еще (не над такси), если необходимо …
end
```

Ясно, что попытку присваивания можно проводить всюду, где может использоваться тест объекта. Новый механизм имеет преимущество не по причине загромождения теста локальными переменными, такими как *t*, – локальная переменная теста играет ту же роль. Но дело в том, что она появляется точно в том месте, где необходимо ее использовать. Помимо этого, использование **Void** в качестве ошибки является небезопасным, так как ничто не заставляет вас выполнять тест на void. Отсутствие проверки приводит к void-вызову со всеми вытекающими последствиями, а при применении теста объекта этого не происходит. В результате честно служившая в течение десятилетий попытка присваивания « подала в отставку», и теперь в стандарте языка Eiffel используется тест объекта.

Динамический кастинг используйте «с умом»

Следует всегда помнить о маячащей угрозе синдрома множества явных вариантов. Динамический кастинг делает возможным реализовать структуру решения в форме: «Если *TAXI*, то делай это, иначе, если *TRAM*, то делай то, иначе, если *BUS* ...».

Детали, кстати, должны быть тщательно учтены – по той причине, что тесты используют *согласование* типов. Если выполняется разбор случаев *MOVING*-объектов, то следует помнить, что, например, объект *TRAM* согласован также с типом *VEHICLE*, так что порядок тестов играет значение.

Все же вы можете делать это.

Ясно, что это не лучшая идея. Слова, поясняющие, почему это плохо, уже сказаны. Динамический кастинг полезен, но не как соперник динамическому связыванию, который побеждает в схватке, когда оба соперника применимы. Динамический кастинг необходим, когда объекты приходят из внешнего мира – файлов, баз данных, сети, чьи типы программа должна анализировать динамически перед их использованием.

В таких случаях часто привидение делается к одному типу, не выполняя проверки всех возможных случаев. Это довольно хороший критерий использования динамического кастинга. Если ожидается некоторый тип объекта и выясняется, что реальность согласована с ожиданиями, то все прекрасно. Если же приходится делать выбор, анализируя весь возможный набор типов, то почти наверняка динамический кастинг используется не по назначению и его место должно занять динамическое связывание.

Есть еще один выход. Если нет свободы в создании новых классов, то применяйте образец «Посетитель», описание которого является заключительной темой этой главы.

16.14. Обращение структуры: посетители и агенты

Дисциплина проектирования, рассматриваемая в этой главе, связана с комбинацией наследования, отложенных классов, полиморфизма, переопределения, динамического связывания, поддерживаемая контрактами, она вырабатывает в результате элегантную и гибкую архитектуру приложения. Но есть и темная сторона, которую нельзя оставить без исследования.

Грязный маленький секрет

«Грязный» секрет (не совсем уж секрет, поскольку коротко о нем упоминалось в начале обсуждения в этой главе) состоит в том, что наш рецепт по сохранению стабильности архитектуры ПО годится только для одного из двух принципиальных вариантов внесения изменений. Применяемые до сих пор приемы программирования обеспечивают гладкую эволюцию программной системы, когда *операции известны*, а изменения связаны с *появлением новых типов*. Но ничего не было сказано о противоположном варианте.

Мне никак не хотелось бы, чтобы вы рассматривали прочитанные десятки страниц этой главы как увертюру к опере, которую предстоит теперь услышать. Эксперименты показывают, что сценарий, глубоко нами изученный – расширение старых операций на новые типы, – наиболее частый в эволюции системы, он требует наиболее тонкого обращения. Здесь проявляется мощь полиморфизма и динамического связывания, демонстрируются успехи объектной технологии.

Но встречается и противоположный сценарий, игнорировать его нельзя.

Предположим, например, что в интересах ряда приложений в систему Traffic следует ввести изменения, позволяющие показ объектов разных видов сопровождать вспышками – повторяющимся несколько раз миганием. Такие вспышки можно разрешить всем объектам или только избранным видам объектов списка. Можно ли добавить эти свойства в ПО, сохраняя ту же степень скрытия информации, в частности, без проверки специфического типа объектов?

Позвольте нам называть **целевыми классами** те классы, в которые добавляются новые свойства, и **клиентскими классами** – классы приложения, которые будут применять новые операции к целевым объектам. В системе Traffic класс *TAXI* может быть примером целевого класса.

В благоприятных ситуациях уже изученные приемы все еще могут успешно применяться.

- Если все целевые классы наследуются от общего предка, то можно добавить новую операцию на верхнем уровне и переопределить ее у потомков нужным образом.
- Если нет общего предка, то можно его создать. Можно ввести, например, класс *FLASHABLE*, а затем, благодаря множественному наследованию, сделать все классы, объекты которых могут мигать, потомками класса *FLASHABLE*.

Эти приемы работают, но они плохо масштабируются, если необходимость в новых операциях возникает довольно часто. Наряду с «миганием» может возникнуть необходимость «вращения», а позже их «подъема» – еще многое, что может придумать человек. Множественное наследование позволяет вводить все новые маленькие классы – *ROTATABLE, RAISABLE* и другие – но все это не выглядит впечатляющим.

> В качестве еще одного важного для практики примера рассмотрим среду разработки, такую как EiffelStudio или Eclipse. В качестве фундаментальной структуры используется абстрактное синтаксическое дерево (АСД), покрывающее целевые классы, такие как *INSTRUCTION, EXPRESSION, LOOP*. Эти классы обладают некоторыми базисными свойствами, но новый клиентский инструментарий непрерывно пополняется: форматизатор программ, анализатор, отыскивающий потенциальные ошибки, генератор HTML – все эти средства могут применять новые операции к каждому узлу АСД, но это встречается довольно редко. В EiffelStudio проблема решается использованием образца «Посетитель», обсуждаемого далее.

В некоторых ситуациях полностью отсутствует возможность модификации целевых классов, например, если они находятся в библиотеке, недоступной для ваших изменений. Тогда предыдущие рецепты не помогают.

Есть две возможности решения такой проблемы.

- Образец «Посетитель» (Visitor).
- Использование механизма агентов.

Схема применения этих приемов состоит в следующем.

Образец «Посетитель» (pattern Visitor)

Образец «Посетитель» позволяет определить произвольные свойства, применимые к экземплярам существующих классов.

Идея очень проста: **пусть операции знают о типах**. Если мы применяем различные операции к различным типам, то либо операция должна знать о типах, либо тип об операциях. При динамическом связывании каждый тип знает о применимых операциях. Теперь будем рассматривать противоположную ситуацию – клиентскому классу необходимо выполнять операцию на экземплярах многих возможных классов (целевых классов). Мы можем называть объекты, такие как такси и трамваи в нашем постоянном примере, *целевыми объектами* или просто *целями*.

Проблема не в том, как определить операцию. Предполагается, что подходящие алгоритмы доступны для написания методов, например

```
flash_taxi (t: TAXI) do … Алгоритм мигания объекта такси …end    [8]
```

Аналогично можно определить метод *flas-tram*, *flash_bus* и так далее.

Вопрос в архитектуре построения – где разместить компоненты и каким способом использовать их, сохраняя расширяемость ПО?

Компоненты не должны принадлежать целевым классам. Если поместить их туда, то динамическое связывание давало бы решение. Но мы полагаем, что это не тот случай; вот почему они должны получать свои цели через аргументы, такие как *t*: *TAXI* в приведенном выше примере программы.

Нет никаких причин требовать и от клиентских классов реализации операций. Лучше всего собрать все методы в отдельный класс, который знает, как выполнять специальную операцию, скажем, *flash*, на различных целях, таких как такси и трамваи.

Так что наш дуэт между клиентом и поставщиком превращается в треугольник между клиентом, поставщиком и посетителем. Посетитель является объектом, способным выполнять одну операцию над многими видами объектов. Ситуация обратная по отношению к динамичекому связыванию, хотя, как доказательство мощи базисных ОО-идей, посетители реализуются как классы и схема реализации существенно основана на динамическом связывании.

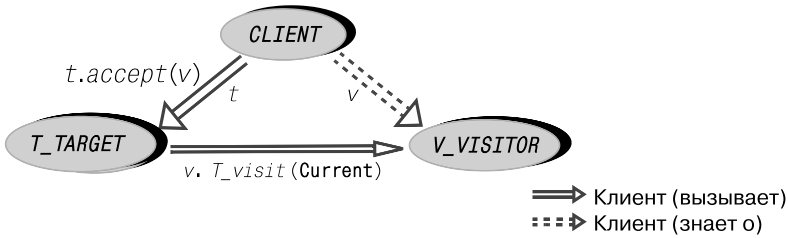


Рис. 16.25. Посетитель (треугольник операций)

Для цели типа *T* и операции *V*, такой как *flash*, рисунок показывает сценарий взаимодействия между:

- целевым классом, *T_TARGET*, задающим целевые объекты типа *T*. Класс *TAXI* является типичным примером;
- классом посетителя, *V_VISITOR*, например, *FLASH_VISITOR*, задающим применение выбранной операции к объектам многих различных типов;

- клиентским классом, задающим элемент приложения, которому необходимо выполнять операции над целевыми объектами различных типов.

Часто клиентскому классу необходимо выполнить операцию на множестве целевых объектов, например, операцию *flash* для всех Traffic объектов из списка. Это объясняет термин «*посетитель*»: посетитель – экземпляр класса, такого как *FLASH_VISITOR,* – позволяет клиенту «посетить» каждый элемент некоторой полиморфной структуры, каждый раз выполняя подходящую версию специфицированной операции. Как вы знаете, процесс выполнения таких визитов называется итерированием, или обходом.

Как всегда, в интересах расширяемости и повторного использования при обсуждении архитектуры ПО важно установить, кому какие требуются знания и кто не должен знать деталей. Цель – уменьшить количество информации, распространяемой по структуре, что особенно важно, когда информация изменяется. Здесь:

- целевой класс *знает о* специфическом типе, таком как *TAXI* (так, например, *TAXI* наследует от *VEHICLE*, а *VEHICLE* от *MOVING*), а также его контекст в иерархии типов. Он *не знает о* новых операциях, запрашиваемых извне, таких как *flash*;
- класс «Посетитель» *знает* все о данной операции и обеспечивает подходящие варианты для релевантных типов, обозначая соответствующие объекты через аргументы. В класс «Посетитель» помещаются методы, такие как *flash_bus, flash_taxi, flash_tram*. Он *ничего не знает* о клиентах;
- клиентскому классу необходимо применять данную операцию к объектам определенного типа, так что он *должен знать типы* (только их существование, но не их свойства) *и операции* (только их существование и применимость к данному типу, но не специфические алгоритмы);
- используя образец «Посетитель», клиент будет способен применить операцию, например, для всех элементов списка без знания индивидуальности типов элементов, как в следующей программе:

```
flash_all (fl: LIST [TARGET]                      –Смотри ниже о TARGET
            – Мигание(flash) всех элементов в fl.
do
            from fl.start until fl.after loop
                  – “Flash fl.item”
                  fl.forth
            end
end
```

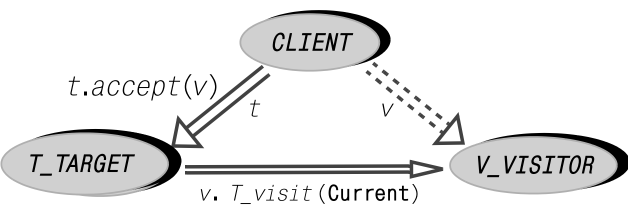


Рис. 16.26. Классы образца «посетитель»

Образец «Посетитель» обеспечивает реализацию, заданную строкой псевдокода. Она проста (следует из рисунка). Базисной операцией посещения является

```
t.accept (v)
```

Здесь *t* – целевой объект, *v* – объект посетителя. Это позволяет заменить строку псевдокода следующим кодом:

```
fl.item.accept (flasher)
```

Здесь *flasher* – это объект *FLASH_VISITOR*.

Все целевые классы должны обеспечить компонент *accept*, общая реализация которого имеет вид:

```
accept (v: VISITOR)
            – Применить релевантную операцию посещения (visit)).
    do
            v. T_visit (Current)
    end
```

Здесь *T_visit* – это метод, реализующий запрашиваемую операцию для типа *T*, например, *bus_visit*, *tram_visit*. Эти методы должны быть реализованы в классе «Посетитель»:

```
bus_visit (t:BUS) do flash_bus (t) end
tram_visit (t:TRAM) do flash_tram (t) end
taxi_visit (i:TAXI) do flash_taxi (t) end
… и так далее …
```

Обычно возможно избежать обертки существующих процедур – вместо этого можно непосредственно реализовать *flash_bus* под именем *bus_visit* и так далее.

Восхищает тонкая хореография дуэта, в которой цель и посетитель помогают друг другу, как только клиент приводит их в движение. Цель знает свой собственный тип, но не знает запрашиваемую операцию, но знает того, кто знает, – это посетитель *v*, переданный клиентом как аргумент *accept*.

Так что цель может вызвать *T_visit* на посетителе, где *T* идентифицирует целевой тип, и передает себя – **Current** – как аргумент. Теперь в игру вступает посетитель, он использует правильную операцию, идентифицируемую включением *T* в имя метода, запуская ее на правильном объекте, идентифицируя его как аргумент, переданный методу.

Хотя образец «Посетитель» предназначен для лечения ограничений динамического связывания, сам он основан на использовании этого механизма фактически дважды:

D1 в вызове клиента *t.accept*(*v*) для выбора правильной цели.

D2 в вызове цели *v. T_visit* (**Current**) для выбора правильной операции (выбирая правильного посетителя).

Динамическое связывание называют также динамической диспетчеризацией, а точнее, одиночной диспетчеризацией, так как правильный алгоритм выбирается (выполняется диспетчеризация) на основе *одного* критерия – типа цели вызова. Образец «Посетитель» называют двойной диспетчеризацией, поскольку здесь выбор осуществляется по двум критериям – по типу объекта и виду операции. В образце иллюстрируется возможная техника реализации двойной диспетчеризации в среде, которая поддерживает только одиночную диспетчеризацию, характерную для большинства ОО-языков и соответствующих сред программирования.

Реализация дважды применяет одиночную диспетчеризацию. Первый вызов, D1, получает цель – объект *t* – и, выполняя одиночную диспетчеризацию, находит правильный для этой цели метод *accept*. В методе с использованием переданной ему в качестве аргумента операции осуществляется второй вызов, D2, где операция выступает в роли цели, а тип в роли аргумента. Здесь снова выполняется одиночная диспетчеризация – динамическое связывание, которое и находит нужный алгоритм, выполняющий требуемую операцию, получая при этом объект нужного типа, переданный как **Current**.

Для того чтобы сработало динамическое связывание при первом вызове D1, все целевые классы должны иметь собственную реализацию метода *accept*, каждый объявляя его в форме *v.T_visit*(**Current**), как показано выше.

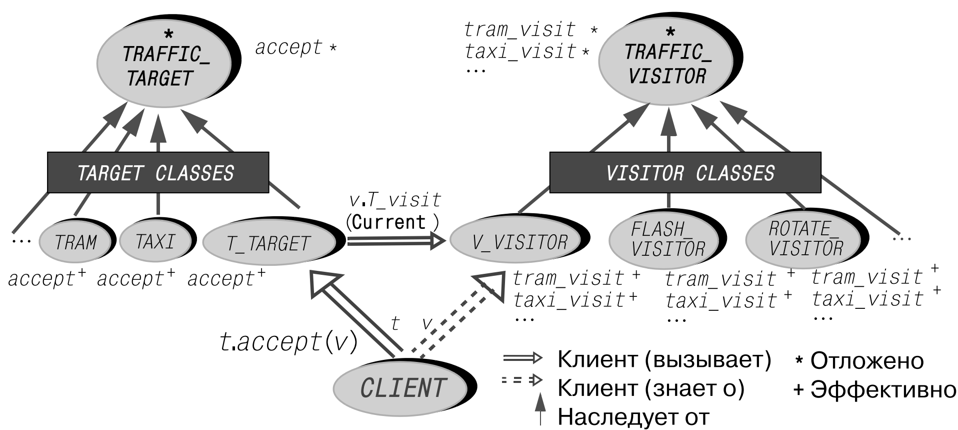


Рис. 16.27. Посетитель (полная схема)

Как показано на рисунке, метод *accept* приходит от общего предка, где он отложен. Этим предком может быть класс *TARGET*, который может быть очень простым:

```
note
    description: "Объекты, которые могут использоваться как цели в образце
                - "Посетитель""
deferred class TARGET feature
    accept (v: VISITOR)
            — Выполнить операцию посещения (visit) на текущем объекте с целью v
            —| Замечание: типичная реализация v.T_visit(Current)
            —| где T - это специфический эффективный тип потомка.
            deferred
            end
end
```

Последние две строчки заголовочного комментария используют стандартное соглашение: если комментарий начинается тройкой символов --|, то это комментарий, задающий свойства реализации и не относящийся к клиентам.

Разочаровывает то, что все целевые классы наследуют от специального класса *TARGET*, поскольку основная идея состояла в том, чтобы использовать целевые классы в исходном виде. С рассмотренными до сих пор приемами программирования, если не предусмотреть такое поведение для целевых классов с самого начала, дело плохо. Это принципиальное ограничение образца «Посетитель». Для его устранения необходимо перейти к технике, полностью отличающейся от образца. Все не так плохо во многих практических случаях. Ведь цель состояла в том, чтобы избежать повторяющейся модификации целевых классов при введении дополнительных операций. Здесь же нужно только убедиться, что целевой класс имеет специального предка с определенной операцией, после чего жизнь становится прекрасной и можно добавлять посетителей по своему усмотрению.

На стороне посетителя также необходим общий предок, назовем его *VISITOR*. С этим требованием проблем не возникает, поскольку все равно нужно создавать специальный класс посетителя всякий раз, когда появляется новая операция.

Предыдущий рисунок показывает все относящиеся к делу классы и отношения наследования между ними. Имена отложенных классов *TARGET* и *VISITOR* теперь начинаются с префикса *TRAFFIC_*, которое следует заменить именем, идентифицирующим приложение в случае создания собственной реализации образца «Посетитель». Эти классы невозможно определить как повторно используемые компоненты. Класс *VISITOR* должен знать, в частности, все целевые типы в приложении, чтобы он мог перечислить даже в отложенной форме релевантные методы, здесь – *tram_visit*, *bus_visit* и так далее.

Как и ранее, *T* и *V* задают в прототипах примеров целевой тип и операцию, дополняемую здесь конкретными классами, такими как *TRAM* и *FLASH*.

Завершая презентацию образца «Посетитель», я хочу надеяться, что вы поняли ее в полном объеме – не только как технику, но и ее точные цели, область возможного применения, принципы, преимущества и ограничения, – и способны применить ее на практике.

Улучшение образца «Посетитель»

Проект образца «Посетитель» представляет популярную и полезную технику, но страдает двумя отмеченными недостатками.

- Для обеспечения «посещаемости» все целевые классы должны быть потомками общего предка. Это не реалистично, если их авторы специально не предусматривали такую цель.
- Образец не задает повторно используемое решение. Он должен заново программироваться при всяком его использовании. Во всех случаях код выглядит похожим, примером является метод *accept*, соответствующий общей схеме.

Используя универсальность, базисную схему можно улучшить. Полное удовлетворительное решение основано на механизме, изучаемом в следующей главе, – **агентах**.

Решение, основанное на агентах, просто в использовании. Для каждого применимого целевого класса *T* пишется метод *visit*, реализующий требуемую операцию для *T*. Это не требует модификации существующего класса или создания нового класса. Затем общему механизму, применяющему операцию к объекту, передается как цель, так и агент *visit* – объект, представляющий операцию. Сам механизм ничего не знает о специфике операций. Упражнение в главе по агентам попросит вас представить решение. Библиотека «образцов», разработанная в ETH, обеспечивает повторно используемый вариант «Посетителя», основанный на этом подходе.

16.15. Дальнейшее чтение

1. Бертран Мейер: «Объектно-ориентированное конструирование программных систем» Изд. Русская Редакция, Интернет-Университет, Москва, 2005 г.
   Содержит детальный анализ наследования на протяжении нескольких глав.
2. Бертран Мейер, Карин Арнаут: «Componentization: the Visitor Example», Computer (IEEEE), vol. 39, no. 7, July, 2006, pages 23-30
   также доступна в интернете:
   se.ethz.ch/`meyer/publications/computer/visitor.pdf

Образец «Посетитель» дополняет динамическое связывание, позволяя просто добавлять операцию в множество существующих типов (в противоположность обращенной задаче). Эта статья представляет образец, предлагая реализующий его повторно используемый компонент, который является частью библиотеки образцов ETH. В литературе можно найти много описаний образца «Посетитель» (включая Википедию), большинство которых используют в качестве источника описание из книги E. GAMMA и др., Design Patterns, Addison-Wesley, 1994.

16.16. Ключевые концепции этой главы

- Наследование означает, что некоторый класс (наследник) получает компоненты и инвариант от другого класса (родителя). Экземпляры наследника могут быть обработаны таким же способом, что и экземпляры родителя.
- Согласование распространяет на типы отношение «быть потомком» между классами.
- Совместно с универсальностью наследование помогает определить изощренную систему типов, которая превращает компилятор в инструментарий, строящий доказательства и проверяющий свойства согласования элементов программной системы.
- Полиморфизм, применяемый только к ссылкам, позволяет выражению, имеющему «статический» тип по объявлению, обозначать объекты разных типов (их динамические типы) во время выполнения программы. Система типов гарантирует, что все динамические типы являются потомками статического типа.
- Форма полиморфизма, основанная на универсальности, позволяет определять контейнерные структуры, которые во время выполнения могут наполняться объектами разных типов.
- Динамическое связывание гарантирует, что в присутствии полиморфизма любой вызов компонента всегда использует версию, наилучшим образом адаптированную к динамическому типу цели.
- Полиморфизм и динамическое связывание используют преимущества скрытия информации, позволяя клиентам игнорировать точный тип объектов, которые они обрабатывают, и не знать, какая именно версия операции была применена. У них нет необходимости в знании этой информации.
- Правила типизации ограничивают полиморфизм: тип любого объекта, к которому может быть присоединена во время выполнения переменная (динамический тип переменной), должен быть согласован с объявленным (статическим) типом. Это требует, чтобы при любом присваивании и передаче аргументов методу тип источника был согласован с типом цели.
- Отложенные компоненты имеют спецификацию, включающую сигнатуру и возможные контракты, но не имеющие реализации. Класс отложен, если он содержит, по

меньшей мере, один отложенный компонент, хотя другие компоненты могут быть эффективными (не быть отложенными). Отложенные компоненты позволяют задавать абстракции высокого уровня и, в частности, полезны при проектировании подходящей таксономии. Нельзя создавать экземпляры отложенных классов.
- Класс может переопределить наследуемый компонент, чтобы обеспечить другую его реализацию, отличающуюся от реализации родителя. Это позволяет комбинировать повторное использование и адаптацию.
- Универсальность и наследование являются дополняющими механизмами для расширения типа. Универсальность обеспечивает параметризацию типов, наследование обеспечивает обобщение и специализацию.
- Ограниченная универсальность делает возможным применять специальные операции к переменным формального универсального типа, требуя, чтобы все соответствующие родовые параметры были согласованы с данным типом, заданным ограничением универсальности.
- Множественное наследование дает классу дополнительные преимущества в результате комбинирования нескольких абстракций. Это простая и эффективная техника. Во избежание неоднозначности любой конфликт имен компонентов должен быть устранен в момент наследования.
- Повторное наследование возникает в результате множественного наследования, когда класс является потомком другого класса более чем по одному пути наследования. Повторно наследуемые компоненты сливаются, если они наследуются под одним и тем же именем, и сохраняются раздельно в противном случае. Любая потенциальная неоднозначность при динамическом связывании разрешается благодаря введению предложения **select**.
- Архитектура программной системы, обеспечивающая гладкую эволюцию, должна строиться так, чтобы минимизировать объем знаний, необходимый каждой части системы для взаимодействия с другими частями.
- Динамическое связывание обеспечивает прекрасное решение эволюции ПО в случае подключения новых типов, имеющих собственные варианты «старых» операций.
- В случае новых операций у «старых» типов широко используемое решение задает образец «Посетитель». Он основан на объектах-«посетителях», действующих в качестве посредников между клиентскими классами и целевыми классами. Возможно построение более общего, полностью повторно используемого решения. Оно основано на механизме агентов.

Новый словарь

Не пугайтесь длине последующего списка, поскольку здесь приводятся вариации, используемые разными авторами и в разных языках программирования. Число фактических концепций не столь велико, например, динамическая диспетчеризация означает то же, что и динамическое связывание. Убедитесь, что вы понимаете все концепции (упр. 16-У.1).

Ancestor	Предок	**Assignment attempt**	Попытка присваивания
Cast	Кастинг, приведение к типу	**Client (of a visit)**	Клиент (визита)
		Conformance	Согласование
Constrai-ned genericity	Ограниченная универсальность	**Deferred feature, class, type**	Отложенный метод, класс, тип
Descendant	Потомок	**Double dispatch**	Двойная диспетчеризация
Downcasting	Приведение вниз		

Dynamic binding	Динамическое связывание	**Dynamic cast**	Динамический кастинг
Dynamic dispatch	Динамическая диспетчеризация	**Effect, effective, effecting**	Эффект, эффективный, эффективизация
Flat view	Плоский облик	**Generic constraint**	Ограничение универсальности
Immediate feature	Непосредственный компонент	**Inheritance**	Наследование
Inherited feature	Наследуемый компонент	**Interface (Java, C#)**	Интерфейс (Java, C#)
		Introduce (a feature)	Введение (компонента)
“Is-a” relation	Отношение «является»	**Multiple inheritance**	Множественное наследование
Name clash	Конфликт имен	**Object test**	Тест объекта
Object-Test local	Переменная теста объекта	**Overriding**	Переопределение
		Parametric polymorphism	Параметрический полиморфизм
Polymorphic expression	Полиморфное выражение	**Polymorphic data structure**	Полиморфная структура данных
Polymorphism	Полиморфизм		
Precursor	Precursor (версия родителя)	**Programs with Holes pattern**	Программы с дырами
Proper ancestor, descendant	Подходящий (правильный) предок, потомок	**Redeclaration**	Переобъявление
		Redefinition	Переопределение
		Refinement	Уточнение
Repeated inheritance	Повторное наследование	**Replication (of a feature)**	Репликация компонента
Routine table	Таблица методов	**RTTI (Run-Time Type Identification)**	RTTI (идентификация типа во время выполнения)
Single dispatch	Одиночная диспетчеризация		
Subclass	Подкласс (субподрядчик)	**Subcontracting**	Субконтракт
		Superclass	Суперкласс (подрядчик)
Taxonomy	Таксономия	**Target (of a visit)**	Цель (визита)
Type narrowing	Сужение типа	**Unconstrained genericity**	Неограниченная универсальность
Virtual table	Виртуальная таблица	**Visitor**	Посетитель (образец)

16-У. Упражнения

16-У.1. Словарь

Дайте точные определения терминам словаря.

16-У.2. Карта концепций

Добавьте новые термины в карту концепций, построенную в предыдущих главах.

16-У.3. Абстрактный синтаксис

Это подготовительное упражнение для следующих двух (написать интерпретатор и компилятор). Цель – построить программы, использующие абстрактный синтаксис, избегая построения парсера. Задача не сложная, но использует приемы, которые не были рассмотрены.

Рассмотрим язык программирования L0 со следующими конструкциями, выраженными здесь неформально с конкретным синтаксисом.

- Единственный тип данных – integer (целые).
- Переменная (целочисленная) имеет имя, заданное произвольной непустой строкой.
- Константами являются отрицательные и положительные целые, в том числе ноль.
- Выражение – это переменная, константа или выражение со знаками операций.
- Знак операции (все операции бинарные) – один из трех: +, -, *.
- Выражение со знаком состоит из двух выражений, соединенных знаком операции, с ожидаемой семантикой (сложение, вычитание, умножение).
- Оператором является один из следующих: skip (пустой), read (чтение), compound (последовательность операторов), assignment (присваивание), conditional (условный), loop (цикл).
- Оператор **skip** не имеет эффекта.
- Оператор **read** *x*, где *x* – имя переменной, читает целое значение от интерактивного пользователя и присваивает его переменной *x*.
- Присваивание записывается в форме *x*:= *e*, где *x* – переменная, *e* – выражение.
- Целое можно использовать в тестах условного оператора и цикла с соглашением, что 0 означает ложь, а любое другое значение – истина.
- Уловный оператор записывается в форме: **if** *e* **then** *i1* **else** *i2* **end**, где *e* – выражение, рассматриваемое как булевское значение, *i1, i2* — операторы.
- Цикл записывается в форме: **from** *i1* **until** *e* **loop** *i2* **end**
- Составной оператор является последовательностью операторов. В конкретном синтаксисе последовательность берется в скобки **do ... end**
- Программа является составным оператором.

1. Используя конкретный синтаксис, напишите программу на L0, которая получает от пользователя два целых и вычисляет наибольший общий делитель – НОД, применяя алгоритм Эвклида и не используя умножение (вы можете написать другие L0-программы как примеры для следующих вопросов).
2. Спроектируйте множество классов – таких как *PROGRAM*, *COMPOUND* и так далее, позволяющих создавать представление абстрактного синтаксиса L0-программ. Используйте наследование подходящим образом. Убедитесь, что классы содержат подходящие процедуры создания, так, чтобы L0-программы были представлены как структуры объектов без конкретного синтаксиса.
3. Напишите программу, которая создает абстрактное синтаксическое дерево (АСТ) для программы вычисления НОД (и любого другого примера, который вы подготовили, выполняя пункт 1).

16-У.4. Разбор АСТ

(Это продолжение предыдущего упражнения. Его задача – создать запись текста программы в конкретном синтаксисе по АСТ. Это задача, обратная к парсингу)

Напишите программу, которая печатает конкретное представление с конкретным синтаксисом, описанным в предыдущем упражнении. L0-программа дается как экземпляр класса *PROGRAM* и ассоциированных объектов (экземпляров *COMPOUND* и так далее). Каждый оператор должен начинаться на новой строке. Для составного оператора предусмотрите вложенность. Ветви условного оператора и тело цикла даются с отступами.

Проверьте результаты вычислений и убедитесь, что программа работает корректно.

16-У.5. Интерпретатор, работающий на абстрактном синтаксисе

(Это продолжение последних четырех упражнений)

Напишите интерпретатор – программу на Eiffel, которая может выполнять любую L0-программу, представленную в виде экземпляра класса PROGRAM и ассоциированных объектов (экземпляров *COMPOUND* и так далее). Семантика L0-такова:

- выполнить задающий программу составной оператор;
- распечатать по одной паре на каждой строке, где число пар определяется числом переменных программы, а печать пары означает печать имени переменной и ее значения, разделенных знаком равенства.

Испытайте ваш интерпретатор на построенных примерах программ.

16-У.6. Компилятор, работающий на абстрактном синтаксисе

(Это продолжение предыдущих упражнений)

Напишите компилятор – программу на Eiffel, которая транслирует любую L0-программу в Eiffel-систему в форме корневого класса и множества вспомогательных классов при необходимости. Проверьте, что результаты работы компилятора и интерпретатора совпадают.

16-У.7. Как много такси

Образцом для этого упражнения может служить функция *pre_taxi_count*. Рассмотрите *fleet*: *LIST*[*VEHICLE*].

1. Напишите функцию, которая вычисляет число экземпляров такси в списке транспортных средств.
2. Напишите функцию, которая вычисляет число прямых экземпляров такси в списке транспортных средств.

16-У.8. Универсальный «Посетитель»

Покажите, как улучшить образец «Посетитель», представляя целевой класс как родовой параметр класса *VISITOR*. Убедитесь, шаг за шагом, что полученное решение находится на том же уровне детализации, что и обсуждение в этой главе. Нужно ли вам все еще множество вариантов *VISITOR?* Является ли решение полностью повторно используемым? Обсудите достоинства и недостатки решения в сравнении с базисным вариантом *VISITOR*.

17

Операции как объекты: агенты и лямбда-исчисление

Объектно-ориентированный каркас уже дал нам множество мощных механизмов для написания наших программ. В этой главе мы снова расширим мощь выражений, добавив механизмы, которые позволяют нам ввести абстрактные операции и возможность передачи их в разные части системы для дальнейшего выполнения.

17.1. За пределами двойственности

Расширение потребует рассмотрения *операций* так, как если бы они были *объектами*. С первого взгляда это противоречит базисной двойственности этих двух понятий.

- Программы манипулируют объектами.
- Они делают это, применяя операции к объектам.

Текстуальная структура наших ОО-программ также основана на этом различии: мы разделяем программы на классы, в основе каждого из которых лежит некоторый тип объектов. Каждая операция присоединяется к классу в виде программы – метода класса.

Кажется, что эти два понятия четко различаются: то, что программа может делать, – это операция, а действия выполняются над объектами.

И все же иногда интересно рассматривать операцию как объект или, более точно, определять объекты, чья единственная роль состоит в описании операции. Мы называем такие объекты агентами. В этой главе они детально рассматриваются, но базисную идею изложить нетрудно прямо сейчас. Простого агента можно получить, используя нотацию

```
agent r
```

Это выражение, его значение – агент, представляющий программу *r*. Поскольку это выражение, то его можно присвоить переменной:

```
a := agent r
```

Здесь переменная *a* должна иметь подходящий тип.

Что мы можем делать с агентами? Понятно, что агент ассоциирован с компонентом (методом), поэтому одно из его использований состоит в вызове этого компонента. После вышеприведенного присваивания *a* обозначает агента, поэтому возможен вызов

```
a.call ([x, y])                                          [1]
```

Эффект от вызова такой же, как и от непосредственного вызова программы *r* для любых применимых *x* и *y*

```
r (x, y)                                                    [2]
```

Метод *call* применим ко всем агентам, он принимает в качестве аргумента единственный кортеж, здесь [*x, y*]. Кортеж – это просто объект, представляющий последовательность значений (надеюсь, вы помните это, но если нет, то прежде чем продолжить чтение, перечитайте раздел 13.5).

Зачем использовать вызов метода *call* с агентом, как в **[1]**, вместо прямого вызова **[2]**? Действительно, если известно, какую процедуру вы хотите вызвать, то пользоваться услугами агента нецелесообразно. Но теперь предположите, что *a* получено от другого программного элемента, например, как аргумент текущего метода. Тогда вы знаете только, что *a* обозначает некий метод (и, как мы увидим далее, какие типы аргументов у этого метода), но не знаем самого метода, в данном примере – *r*.

Агенты дают нам возможность построить и поставлять объекты, представляющие операции, готовые к выполнению, с полным разделением между:

- *определением* агента – точкой ПО, где агент связывается с известной программой *r*, выполняя **agent** *r*;
- *вызовом* агента *call* – любой точкой ПО, которая получает агента *a* и может применить *call* для вызова агента, не зная точно, какую программу он принесет.

Механизм имеет много различных применений. В данный момент проанализируем несколько наиболее важных.

- *Итерация*: обеспечение общего механизма, который применяет произвольную операцию к каждому элементу структуры данных.
- *Численное программирование*: подынтегральная функция может рассматриваться как агент при написании метода вычисления определенного интеграла на заданном интервале.
- *Поставка интерактивного приложения с механизмом отката: undo-redo*.

Еще одна область, где агенты играют важную роль, – это программирование, управляемое событиями: образец, известный под именем «Писатель-Читатели» или «Издатель-Подписчики», полезный, в частности, для реализации GUI (графического интерфейса пользователя), являющегося предметом следующей главы.

Что нам предстоит изучить? Сравним агенты с другими приемами, основанными на ранее изученных механизмах, в частности, с динамическим связыванием, также применимым к некоторым из рассматриваемых применений агентов. Дадим краткий обзор математических основ – восхитительной теории лямбда-исчисления. Проэкзаменуем некоторые приемы, которые доступны в языках, отличных от Eiffel.

17.2. Зачем операции превращать в объекты?

Для начала хорошо бы понять, зачем нам нужно рассматривать операции как объекты и что было бы в отсутствии такой возможности. Рассмотрим четыре примера: итерацию, интегрирование, наблюдение, откат.

Четыре приложения агентов

Начнем с **итерации**. Нам приходилось использовать в циклах схему, где операция применяется к каждому элементу последовательной структуры, такой как список. Схема выглядит примерно так:

```
from start until after loop
    " Применить действие к элементу "                                    [3]
    forth
end
```

Здесь *start* устанавливает курсор на первом элементе, *forth* перемещает его к следующему элементу структуры, *after* позволяет выяснить, достигнут ли конец структуры, а *item* позволяет получить элемент, заданный курсором.

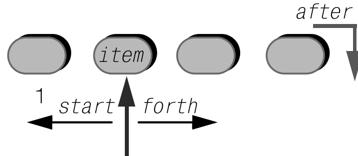


Рис. 17.1. Операции над списком

В Traffic эту схему можно применять к экземпляру *ROUTE*, обозначающему маршрут с остановками. Возможно, нам захочется распечатать список всех остановок в порядке следования их на данном маршруте. Возможно, нам необходимо подсчитать общее время прохождения маршрута (зная время, затрачиваемое на проезд от одной остановки до следующей). Возможна и такая экзотическая операция, как получение списка всех близлежащих ресторанов, зная, какие рестораны находятся в окрестности каждой станции. Для этих и многих других случаев общее решение соответствует схеме **[3].** У нас есть имя для таких схем – итерация (или итерирование), уже встречавшееся при обсуждении структур данных. Можно использовать такую схему для любого *действия*, имея программу, выполняющую *действие* для каждой остановки вдоль маршрута.

Теперь предположим, что по лености, свойственной программистам, или помня, что дублирование кода – это признак плохого программирования, вы не хотите каждый раз писать новый метод, реализующий схему для нового действия. Можно ли подняться вверх по шкале абстракции и написать метод, подобный **[3],** где действие будет выступать в роли аргумента? Тогда мы могли бы применять этот метод для различных действий, возлагая на него заботу об итерировании.

Механизм итерирования на самом деле обеспечивает нам такой метод, названный *do_all*, которому при вызове передается агент, задающий действие:

```
your_route.do_all (agent action)
```

Наш второй пример из области вычислительной математики: **интегрирование**. Дана функция *f*(*x: REAL*): *REAL*. Функция определена на интервале [*a*, *b*]. Существует численный алгоритм (ниже будет дано его обоснование), позволяющий получить хорошую аппроксимацию интеграла от функции *f* на заданном интервале:

$$\int_a^b f\,(x)\,dx$$

Проблема в том, чтобы определить в классе *INTEGRATOR* общий механизм интегрирования – метод, названный, например, *integral*, допускающий повторное использование, чтобы

его можно было бы применять для программ, задающих реализации различных математических функций. Агенты позволяют решить эту проблему, обеспечивая нужный механизм:

```
your_integrator.integral(agent f, a, b)
```

Здесь *your_integrator* имеет тип *INTEGRATOR*.

Третий пример предваряет рассмотрение следующей главы: программирование, управляемое событиями. Предположим, что в некоторых частях программной системы могут возникать в процессе выполнения «события», а другие части системы в ответ на события должны выполнять определенные действия. Примером события может служить событие *tick* системных часов, уведомляющее об истечении очередного кванта времени. В ответ на это один модуль должен изменить визуальный образ часов, отображаемых на экране дисплея, другой модуль должен обновить свой счетчик времени и так далее. Каждый такой модуль – «подписчик» события – должен зарегистрировать некоторое действие, выполняемое всякий раз, когда происходит событие. Спроектировать архитектуру, позволяющую подписчикам достигать требуемого эффекта, достаточно просто при использовании агентов:

```
clock_tick.subscribe (agent some_routine)
```

Здесь *clock_tick* представляет тип события, а *subscribe* – общецелевой библиотечный метод. О таких подписчиках говорят, что они «наблюдают» за событиями определенного типа.

Последний рассматриваемый нами пример соответствует функциональности, необходимой, как правило, любым интерактивным системам – операции «отката». Никогда не доводилось видеть памятник, установленный изобретателю функциональности, которая возникает при нажатии простой пары клавиш CTRL-Z, а ведь возможность отката – это один из краеугольных камней в истории человечества, учитывающий нашу необходимость обезопасить себя от собственных ошибок. Даже если и не иметь в виду путаницу, являющуюся следствием наших действий, часто просто хочется проверить некоторую идею, а затем вернуться назад, если полученный результат не устраивает.

По-настоящему хороший механизм *undo-redo* – это тот, который позволяет отменить не только последнюю операцию, но многие. Пользователю ПО не нужно долго пояснять, *что* делает механизм, но вот *как* он это делает, как самому реализовать программу со встроенным механизмом *undo-redo* – задача более сложная.

Наиболее радикальный способ включает представление всех действий, реализующих откаты и повторы, как объектов, которые можно поместить в структуру данных, скажем, *history*, которая может быть реализована как список из пар агентов:

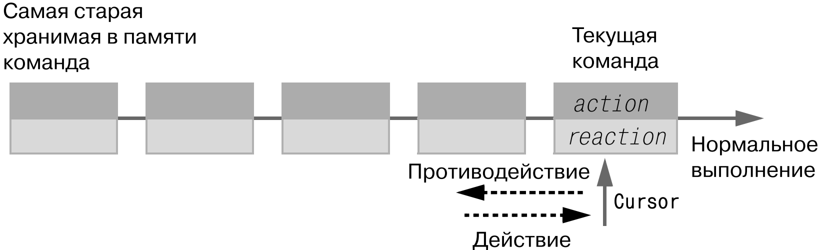


Рис. 17.2. Список истории

Каждое действие выполняется некоторым методом, назовем его *r*. В этом решении система никогда не вызывает непосредственно сам метод *r* для выполнения очередного действия. Вместо этого вызывается

```
execute (agent r, agent r_inverse )
```

Здесь *execute* выполняет вызов *call* (механизм вызова метода, ассоциированного с агентом), используя первый аргумент, но также пару объектов, переданных как аргументы, записывает в список истории. Каждая пара в списке истории содержит два агента, один представляет действие, другой противодействие, – метод, отменяющий результат действия. Предполагается, что всякий раз, когда вы создаете метод *r,* реализующий некоторую команду, одновременно создается и метод *r_inverse*, отменяющий действие (в противном случае реализовать механизм откатов и повторов не удалось бы). Так что если пользователь запрашивает откат на несколько шагов, необходимо выполнить

```
history.item.reaction.call ([])
history.back
```

Эта пара операторов вызывается несколько раз, по числу шагов отката, но, конечно, не далее начала списка истории. Для повтора, запрашиваемого после одного или более отката, выполняется

```
history.forth
history.item.action.call ([])
```

Опять-таки повторы выполняются не далее последнего элемента.

Мир без агентов

Нельзя понять по-настоящему агентов, если не задаться вопросом: а как пришлось бы действовать в случае отсутствия подобного механизма?

Можно ли вообще найти решение? Конечно, можно. Если то, что вам нужно, задается объектом, представляющим обертку действия, то достаточно создать этот объект привычным способом, задав нужный класс. Единственная преграда – пришлось бы создавать большое число новых классов. Давайте посмотрим, как эта идея будет работать на предыдущих примерах.

Интегрирование типичный случай. При определении функции, выполняющей интегрирование, *integral*, введем аргумент, задающий подынтегральную функцию, типа *INTEGRATABLE_FUNCTION*. Соответствующий класс будет отложенным классом, который может выглядеть так:

```
note
    description: " Функции, допускающие интегрирование на конечном интервале"
deferred class INTEGRATABLE_FUNCTION feature
    item (x: REAL): REAL
                                        – Значение функции в точке x.
```

```
            deferred
            end
end
```

Можно спроектировать более изощренную форму этого класса, например, добавив запрос *defined*(*x*: *REAL*) и использовать его в предусловии *item*, но этой простой версии достаточно для понимания архитектурных проблем.

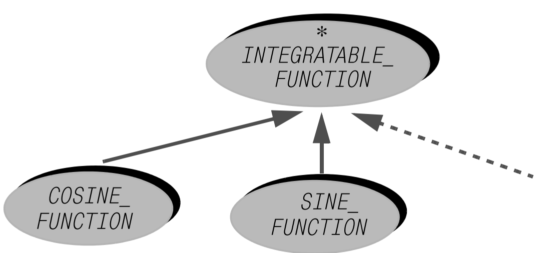


Рис. 17.3. Классы для математических функций

Для каждой функции, которую необходимо проинтегрировать, необходимо, как показано на рисунке, создать небольшой эффективный класс, такой как *COSINE_FUNCTION*, который обеспечивает требуемую реализацию отложенного метода *item*:

```
            – В классе COSINE_FUNCTION
item (x: REAL): REAL
            – Значение функции в точке x.
    do
            Result := cosine (x)
    end
```

Тогда для получения интеграла от функции cos(x) на интервале [a,b] необходимо объявить переменную *f*: *INTEGRATABLE_FUNCTION,* убедиться, что динамически она присоединена к объекту типа *COSINE_FUNCTION*, и вызвать

```
your_integrator.integral(f)                                    [4]
```

Функцию *integral*(*f*: *INTEGRATABLE_FUNCTION*) написать несложно, используя любой алгоритм численного вычисления интегралов. Всякий раз, когда необходимо вычислить значение функции в некоторой точке *x*, используется вызов *f.item*(*x*). Заметьте роль динамического связывания: тип *f* во время выполнения, такой как *COSINE_FUNCTION*, определяет, какая подынтегральная функция (*item*) будет использована. Чтобы убедиться, что вы понимаете схему и фундаментальные ОО-приемы программирования, – хорошая идея написать *integral* самостоятельно.

Время программирования!

Библиотека интегрирования без агентов

Напишите класс *INTEGRATOR* с методом *integral*, который вычисляет интеграл на конечном интервале от функции, передаваемой в качестве аргумента типа *INTEGRATABLE_FUNCTION*. Спроектируйте подходящих потомков *INTEGRATABLE_FUNCTION*, примените вашу работу для вычисления интегралов различных функций. В качестве простого алгоритма интегрирования можно использовать модель, описанную ниже, для версии интегрирования, основанной на агентах.[1]

Этот пример типичен и демонстрирует преимущества стандартной ОО-техники, которую можно использовать, если не иметь агентов. Идеи достаточно просто распространяются и на остальные наши примеры.

- Для итерации можно создать отложенный класс *ITERATABLE_ACTION*, эффективные потомки которого обеспечат специфическую версию процедуры *call*, описывающей выполнение итерируемой операции.
- Для наблюдателей события (программирование, управляемое событиями) отложенным классом является класс *OBSERVER*, потомки которого и обеспечивают свою версию метода *update*, вызываемую издателем при возникновении события. Хорошо известный образец «Наблюдатель» (*OBSERVER)* будет обсуждаться в следующей главе.
- Для откатов-повторов (*Undo-Redo*) для каждой команды интерактивной системы следует создать класс с двумя методами: *execute*, выполняющий команду, и *cancel*, выполняющий откат, – устраняя эффект последнего выполнения *execute*. Экземпляр этого класса описывает информацию, появляющуюся в результате однократного выполнения команды, и необходимый откат, выполняемый в случае запроса. Например, в текстовом редакторе экземпляр *LINE_DELETION* имеет два поля: содержимое строки перед удалением и позицию этой строки в тексте, – так что метод *cancel* сможет восстановить строку, удаленную при выполнении команды *execute*. Все такие командные классы наследуют от отложенного класса *COMMAND*, в котором *execute* и *cancel* являются отложенными методами. Список истории может быть реализован, например, как *LINKED_LIST*[*COMMAND*]. Это схема еще одного классического образца проектирования – *COMMAND.*

Мы можем называть эти приемы образцом проектирования «много маленьких оберток», поскольку здесь используется динамическое связывание, основанное на написании классов, типично небольших, представляющих обертку соответствующей операции.

Образец работает, но имеет очевидный недостаток, о чем свидетельствует данное ему имя, – наполнение ПО большим числом небольших классов. Вообще-то, в небольших классах нет ничего ошибочного. Но принципиально класс должен задавать важную абстракцию, так что кажется подозрительным класс, у которого только один важный компонент. Это подозрение усиливается наблюдением, что в двух приведенных примерах (интегрирование и итерирование) у классов кроме одного важного метода (*item*, *call*) нет атрибутов, а следовательно, нужен только один экземпляр класса, такой как экземпляр

[1] Можно построить и более простую версию, рассматривая отложенный класс INTEGRATOR как «программу с дырами». Вместо того чтобы вводить отдельный класс INTEGRATABLE_FUNCTION, достаточно в классе INTEGRATOR ввести отложенный метод integrated_function – аналог item. Потомки класса будут переопределять эту функцию, задавая таким образом нужную им подынтегральную функцию. В этом варианте у метода integral нет необходимости вводить аргумент f.

COSINE_FUNCTION, присоединенный к *f* в **[4].** Класс с одним экземпляром известен как «*Одиночка*» (*Singleton*). Но здесь объект не только присутствует в единственном экземпляре, но у него и полей вообще нет – странный объект, в самом деле. Каждый из классов представляет капсулу с одной единственной процедурой. Мы можем называть такие классы «певец одной песни».

Написание многих таких оберток усложняет ПО, в частности, структуру наследования, как мы увидим в следующей главе при рассмотрении образца «Наблюдатель». В конце концов, это ужасно, что нельзя непосредственно использовать функцию *cosinus* для интегрирования или программу *print_stop_name* при обходе маршрута. Зачем нужно обертывать ее в одежды класса? В экстремальных случаях одна и та же математическая операция могла бы применяться для интегрирования, итерирования и наблюдения. Неужели ей нужны три разных наряда?

Среди наших примеров один из них – откаты и повторы – не кажется с этих позиций искусственным, поскольку абстракция *COMMAND* обоснована. У нее есть два важных метода – *execute* и *cancel*, так что у певца есть, по крайней мере, две песни, есть атрибуты, так что потомки описывают имеющие смысл различающиеся экземпляры.

В других наших примерах трудно согласиться с техникой «много маленьких оберток». Нам нужно обертывать операции, превращая их в объекты, но мы не хотим делать это самостоятельно. Лучше возложить это на некоторый механизм, встроенный в язык.

Таким механизмом являются агенты. Если *f* – это метод, то нетрудно получить в качестве подарка этот метод в красивых одеждах, просто написав **agent** *f*. Мы получим объект, обладающий всем, что есть у *f*, включая возможность вызова *f* (используя *call*) для любых применимых аргументов, который можно вызывать всюду и всякий раз, когда это понадобится. Во всех рассмотренных случаях и во многих других агенты являются соперниками образца «много маленьких оберток». Надеюсь, маленькое отступление – как обойтись без агентов – дает нам лучшее понимание преимуществ простого, встроенного механизма, позволяющего работать с действиями, как с объектами.

17.3. Агенты для итерации

Теперь, когда мы видели, где полезны агенты и почему они нам необходимы, мы можем выйти за границы предыдущего рассмотрения и заняться полной картиной со всеми деталями. Данный раздел дополняет пример с итерированием, следующий – интегрированием. В следующей главе будет представлено детальное описание образца «Наблюдатель», основанное на агентах.

Базисные схемы итерирования

Простой пример из Traffic иллюстрирует использование и определение итератора через агенты. Рассмотрим понятие маршрута *ROUTE*. Мы можем добавить в *ROUTE* метод *do_at_every_stop*, который принимает действие как аргумент и применяет его к каждой остановке. Это делает возможным вызовы:

```
your_route.do_at_every_stop (agent print_stop_name)                    [5]
your_route.do_at_every_stop (agent append_restaurants)
...
your_route.do_at_every_stop (agent other_operation)
```

Предполагается, что *print_stop_name*, *append_restaurants, other_operations* являются методами, принимающими *STOP* в качестве аргумента.

Как должен выглядеть метод *do_at_every_stop,* чтобы все эти вызовы стали возможными? Он абстрагирует стандартную схему итерации, приведенную в этой главе [3]:

```
do_at_every_stop (action :…)
            – Применить действие к каждой остановке данного маршрута.
   do
            from start until after loop
                    action.call ([item])                                     [6]
                    forth
            end
end
```

Для включения ассоциированного метода используется вызов *call* – процедура, доступная всем агентам, чей эффект состоит в вызове метода агента с заданными аргументами; более точно, аргументом является единственный кортеж, здесь [*item*]. Как вы знаете, последовательность значений, заключенная в квадратные скобки, представляет манифестный кортеж[1]. Так как предполагается, что *action* представляет метод, такой как *print_stop_name*, у которого один аргумент, кортеж, используемый здесь, [*item*], имеет ровно один элемент.

Эффект от вызова *action.call*([*item*]) **[6]** точно такой же, как и при прямом вызове соответствующего метода, такого как

```
print_stop_name (item)                                                       [7]
```

Это справедливо при условии, что аргумент, переданный *do_at_every_stop*, был **agent** *print_stop_name.* Если же аргумент – **agent** *append_restaurants,* то это соответствует вызову

```
append_restaurants (item)                                                    [8]
```

Разница с **[6]** в том, что внутри *do_at_every_stop* мы ничего не знаем, какой фактический метод задает *action*.

Эта техника является базисным механизмом итераторов в библиотеке EiffelBase. Далее мы, фактически, будем изучать библиотечную реализацию.

Итерирование для исчисления предикатов

Интересное приложение итерирования состоит в непосредственной реализации механизмов исчисления предикатов: кванторов «Для всех» (∀) и «Существует» (∃). Предположим, что вы хотите установить, что все элементы массива целых *a* в границах *a.lover* и *a.upper* являются положительными. В исчислении предикатов это выражается записью:

```
∀ s: a.lower .. a.upper | a [i] > 0                                          [9]
```

[1] В Eiffel манифестными называются неименованные константы, заданные своими значениями, такие, как 17 или «это константа». Тип манифестных констант однозначно определяется синтаксисом их записи. Манифестный кортеж [item] однозначно позволяет установить, что это константа, задающая кортеж с одним элементом.

Без агентов вы могли бы использовать *all_positive*(*a*), написав функцию *all_positive*(*ia: ARRAY*[*INTEGER*]), получающую результат, выполняя цикл. С агентами нет необходимости в такой функции, можно написать проще:

```
(a.lower |..| a.upper).for_all (agent is_positive)                    [10]
```

Такая запись по стилю близка к предикатной записи [9]. Символика |..| является операторным псевдонимом (alias) функции *interval* из класса *INTEGER*. Результат выполнения данной функции принадлежит типу *INTEGER_INTERVAL* – библиотечному классу, содержащему функции *for_ all* и *exist*. Данные конкретные функции *for_ all* и *exist* применимы к массивам, но вскоре мы познакомимся с аналогичными функциями, применимыми к спискам и другим последовательным структурам.

В [10] все же требуется проверка на положительность и запрос *is_positive(n:INTEGER): BOOLEAN*. Это более разумно, чем требовать нечто подобное *all_positive* для каждого такого случая. В конце главы мы научимся, как избавиться даже от *is_positive*, написав нужный агент без введения явной процедуры.

Время проведения исследований!

Есть смысл теперь взглянуть на функции *for_ all* и *exist*, как они заданы в классе *INTEGER_INTERVAL.* Пусть вас не смущает объявление типов (о них мы еще поговорим), но убедитесь, что вы понимаете реализацию. Посмотрите также и на функцию *exist1*.

Типы агентов

В объявлении *do_at_every_stop* необходимо указать тип *action*, представляющий агента. Фактическое объявление выглядит так

```
do_at_every_stop (action: PROCEDURE[ANY, TUPLE [G]]
                … Остальное как ранее [6]…
```

Универсальный библиотечный класс *PROCEDURE* описывает агента – связанную с ним команду (процедуру). У класса два родовых параметра, представляющих типы, которые характеризуют процедуру *p*, связанную с агентом.

- Первый обозначает класс, от которого приходит *p*, или предка этого класса. Так как *ANY* является предком всех классов, можно обычно использовать *ANY*, как это сделано здесь для *do_at_every_stop*, поскольку фактический параметр **agent** *P*, соответствующий *action,* не использует информацию о том, какому классу принадлежит *P*.
- Второй аргумент – это всегда кортеж. Типы аргументов процедуры *P* должны быть согласованы с типами компонентов кортежа.

Параметр *P* должен быть процедурой, такой как *print_stop_name* или *append_restaurants.* У нее один аргумент типа *G* (родовой параметр *LINEAR* и его потомков, который также служит как тип для *item* и представляет тип элементов структуры данных). Как следствие, второй родовой параметр *PROCEDURE* должен быть *TUPLE*[*G*].

Выбор *TUPLE*[*G*] в качестве второго параметра – это то, что позволяет в теле *do_at_every_stop* вызывать *P*, какой бы она ни была, с правильными аргументами, используя такую запись из [6]

```
action.call ([item])                                              [11]
```

Фактически метод *call* объявлен в классе *PROCEDURE*, принимая аргумент типа *OPEN* – второй родовой параметр.

Класс *PROCEDURE* описывает агентов, связанных с командами. Он является частью иерархии из четырех классов в библиотеке KERNEL:

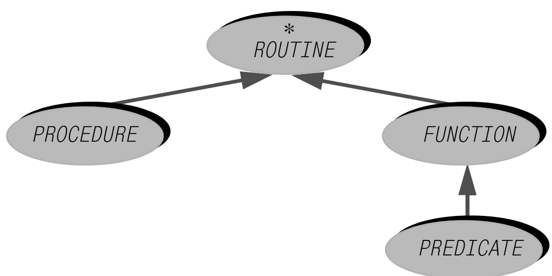


Рис. 17.4. Классы агентов

Класс *FUNCTION* предназначен для агентов, обозначающих запросы, за исключением запросов, возвращающих результат типа *BOOLEAN*, последние покрываются классом *PREDICATE*. Класс *FUNCTION* имеет третий родовой параметр, задающий тип результата функции. Класс *ROUTINE* – отложенный класс покрывает все варианты агентов.

Приведу заголовки данных классов:

```
deferred class ROUTINE [BASE, OPEN -> TUPLE]
class PROCEDURE [BASE, OPEN -> TUPLE] inherit
   ROUTINE [BASE, OPEN]
class FUNCTION [BASE, OPEN -> TUPLE, RES] inherit
   ROUTINE [BASE, OPEN]
class PREDICATE [BASE, OPEN -> TUPLE] inherit
   FUNCTION [BASE, OPEN, BOOLEAN ]
```

Второй параметр *OPEN* ограничен *TUPLE*, так что можно использовать только кортежный тип *TUPLE*[*G*] как фактический параметр. Выражение **agent** *f* в зависимости от природы *f* принадлежит одному из вышеприведенных типов – процедура, булевский запрос или запрос другого типа.

Для агентов, представляющих процедуры с двумя аргументами типов T и U, используйте

```
PROCEDURE [C, TUPLE [T, U ]]          – в качестве C – часто задается ANY
```

Для программ без аргументов второй фактический параметр будет просто *TUPLE*, как в *PROCEDURE*[*ANY, TUPLE*]/

Класс *ROUTINE* объявляет:

```
call (v: OPEN)
   – Вызов компонента с операндами, используя v для открытых операндов.
```

Это позволяет вызывать агента, передавая подходящий кортеж, как показано выше [11]. Если здесь нет аргументов – фактический параметр для *OPEN* просто *TUPLE*, – *call* передается пустой кортеж.

Помимо прочего, *FUNCTION* и *PREDICATE* включают запрос:

```
last_result: RES
– Возвращается результат последнего вызова call, если он есть.
```

Для удобства эти классы включают функцию *item*, комбинирующую вызовы *call* и *last_result*:

```
item (v: like open_operands): RES
      – Результат вызова компонента со всеми операндами, используя v для
      – открытых операндов.
      – (Будет вызывать call.)
   ensure
      set_by_call: Result = last_result
```

Таким образом *f.item*([*x*]) для агента *f* дает результат вызова ассоциированной функции с аргументом *x*.

Дом для фундаментальных итераторов

Класс *LINEAR* в EiffelBase – предок всех списковых классов, таких как *LIST*, *LINKED_LIST* и других, описывающих любую структуру, которую можно обойти линейно. Как таковой, он является естественным домом для множества компонент, представляющих итераторы:

- *do_all* применяет некоторое действие поочередно ко всем элементам структуры, подобно *do_at_every_stop*;
- *do_if* применяется ко всем элементам, удовлетворяющим некоторому условию. Вариантами являются *do_while* и *do_until*;
- *for_all* – тест проверки выполнения некоторого условия (представленного агентом) на всех элементах структуры. Аналогично *exists* проверяет выполнение условия хотя бы на одном элементе.

Аргументами являются:

- в первых двух категориях – *action*, представляющее применяемое действие, типа *PROCEDURE*[*ANY, TUPLE*[*G*]];
- в последних двух категориях – *test*, представляющий булевский запрос типа *PREDICATE*[*ANY, TUPLE*[*G*]].

Итераторы *do_if, do_while* и *do_until* имеют оба аргумента. В качестве примера использования рассмотрим некоторый класс, имеющий целочисленный атрибут *sum* и процедуру:

```
increase_sum (n: INTEGER)
           – Добавить n к sum.
      do sum := sum + n
           ensure added: sum = old sum + n
      end
```

Зададим список *il: LIST[INTEGER]*. Для этого списка после выполнения присваивания (*sum*:= 0) можно найти сумму всех элементов, вызвав:

```
il.do_all (agent increase_sum)
```

Написание итератора

Конечно, я не сомневаюсь в вашей способности использовать итераторы, такие как *do_all*, но мы должны научиться создавать их. Давайте обратимся к внутренней картине.

Урок анатомии

Мы уже привыкли к анализу реального промышленного кода. Давайте проанализируем код *do_all* из класса *LINEAR*[*G*] библиотеки EiffelBase. Заодно разумно просмотреть код *do_if* и других итераторов.

Вот код, скопированный из библиотеки[1]:

```
do_all (action: PROCEDURE [ANY, TUPLE [G]])
            -- Применить action к каждому элементу.
            -- Семантика не гарантируется, если action изменяет саму структуру;
            -- В таких случаях применяйте итератор не к структуре, а к ее клону.
      local
            c: CURSOR
      do
            c := cursor
            from              -- Основной цикл
                  start
            until
                  after
            loop
                  action.call ([item])
                  forth
            end
            go_to (c)
end
```

Мы можем проигнорировать операторы, связанные с курсором, и сосредоточиться на сигнатуре, комментариях и части, помеченной как «Основной цикл».

Процедура *do_all* принимает единственный аргумент – *action,* представляющий агента, который будет многократно выполняться. Его тип *PROCEDURE*[*ANY, TUPLE*[*G*]] указывает, что процедура, связанная с агентом, может приходить от произвольного класса (поскольку указан класс *ANY*) и (поскольку указан *TUPLE*[*G*]) должна принимать один аргумент типа G – формальный родовой параметр охватывающего класса.

Заголовочный комментарий предупреждает, что «семантика не гарантируется, если действие изменяет структуру». Предупреждение важно: может возникнуть хаос, если действие изменяет саму *структуру*, добавляя или удаляя, например, элементы (в упражнении вас попросят проверить такую ситуацию, но нужно иметь крепкие нервы). Вполне допустимо изменять *содержимое* элементов структуры в результате выполнения действия. Например, вполне безопасно использовать *do_all* для добавления 1 к каждому элементу списка целых:

[1] Комментарии, естественно, изменены.

```
do_all (agent increment)
```

Процедура *increment* изменяет элементы, но не модифицирует структуру:

```
increment(item : INTEGER)
—Увеличивает элемент на 1
   do
      item = item + 1
   end
```

Если требуется модифицировать структуру, то, как указано в комментарии, итерирование безопасно выполнять на клоне исходной структуры. Тогда действие *action* может модифицировать исходную структуру, не воздействуя на ее клон. Для клонирования структуры *s* достаточно выполнить *s.cloned*.

> Требование, устанавливаемое заголовочным комментарием, вполне законно. Понятие итерирования структуры данных становится бессмысленным, если структура изменяется в процессе итерирования. Все же, с сожалением следует отметить, что это важное требование задается в заголовочном комментарии, а не контракте – предусловии *action.* Контракты в настоящее время не обладают выразительной силой, позволяющей задать подобные свойства.

«Основной цикл» — сердце алгоритма – использует ту же схему, что и специальный случай **[6]:**

```
from start until after loop
      action.call ([item])                                        [12]
      forth
end
```

Эти пояснения помогают понять основные свойства *do_all* и подобных операций.

Так как мы изучаем ПО в том виде, как оно представлено в библиотеках, полезно пойти немного дальше в изучении деталей реализации, чем это обычно делается, комментируя производительность и поясняя, как алгоритм работает с курсором.

Прежде всего, хотя вы уже могли догадаться, главная оптимизация компилятора, являющаяся основой успеха приведенной схемы, связана с оператором, помеченным **[12].** Манифестный кортеж, такой как [*item*], представляет объект класса *TUPLE*. После первой его передачи методу *call* объект более не нужен, но наивная реализация создавала бы его на каждом шаге цикла. Это плохо с позиций производительности, не только из-за проблем с памятью — в конечном итоге сборщик мусора утилизирует ненужные объекты, но из-за потерь времени, поскольку создание объекта – дорогая операция. Чтобы избежать потерь, следует, подобно человеку, который жить не может без чашечки кофе, но не использует бумажные одноразовые стаканчики, а приобретает красивую, удобную чашку, создать единственный объект для кортежа и повторно его использовать.

Эту оптимизацию можно запрограммировать явно, объявив соответствующую переменную *t*, создать до начала цикла кортеж [*item*], присвоить его *t* и передавать *t* вместо манифестного кортежа [*item*] как аргумент *call*.

На самом деле заботиться об этом не нужно, поскольку эту оптимизацию выполняет компилятор.

Почувствуй оптимизацию

Повторное использование кортежа

В схемах, таких как [12] для *do_all*, требующих создания многих кортежей, каждый из которых используется однократно, компилятор EiffelStudio генерирует код, который создает единственный кортеж и многократно его использует.

Последней рассматриваемой деталью кода *do_all* является переменная *c*, связанная с курсором. Ее назначение – гарантировать, что итератор оставляет структуру в том состоянии, в котором он ее нашел.

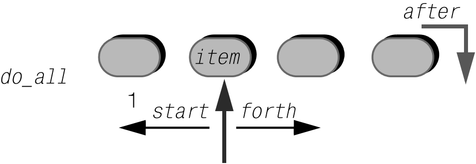


Рис. 17.5. Список с курсором

Структуры *LINEAR* имеют курсор. Итерирование в *do_all* передвигает его, используя *start* и *forth*. Но другие части ПО также могут работать с этим списком и, передвинув курсор к определенной позиции, могут рассчитывать, что он там и окажется при следующем обращении к списку. Итераторы, такие как *do_all*, должны быть «законопослушными гражданами», они могут изменять положение курсора во время выполнения собственных операций, но в конце они должны возвратить его на прежнее место в позицию, представленную переменной *c*.

Экземпляр *CURSOR* является «внешним курсором», который представляет позицию в структуре, допускающей обход, но в отличие от «внутреннего курсора» хранится не в самой структуре, а как внешний объект. В данном случае мы используем внешний курсор для запоминания начальной позиции внутреннего курсора и восстановления позиции при выходе.

17.4. Агенты для численного интегрирования

Как показало рассмотрение итераторов, агенты позволяют нам описать операции, манипулирующие другими операциями. Такие же потребности часто возникают в вычислительной математике: вычисление интегралов – типичный пример.

Стандартная техника численного вычисления интеграла от вещественной функции *f* на конечном интервале *a...b* состоит, как упоминалось в предыдущей главе, в аппроксимации точного значения интеграла ФОРМУЛА!!!! конечной суммой площадей многих маленьких прямоугольников.

Если все эти прямоугольники имеют ширину step, то прямоугольник с координатами по оси абсцисс (*x, x* + *step*) имеет площадь *step * f(x)*. Аппроксимация интеграла на заданном интервале является суммой

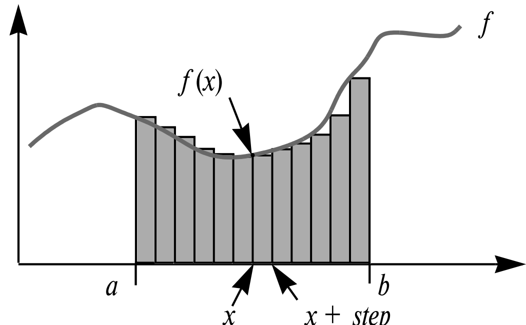


Рис. 17.6. Численное вычисление интеграла

$$\sum_{i=0}^{n-1} f(a + i \times step) \times step$$

всех площадей для всех *x*, таких, что $a \leqslant x <. b$. Число прямоугольников примерно равно $(b - a)/ step$.

Агенты дают нам возможность написать функцию вычисления интеграла *integral* не только для конкретной подынтегральной функции *f* – например, функции косинус (*cosine*), – но и для любой применимой функции.

Вот пример реализации *integral*:

```
integral (f : FUNCTION [ANY, TUPLE [REAL], REAL]; a, b : REAL): REAL
            – Аппроксимация вычисления интеграла от функции f на интервале a.. b
    local
            x: REAL ; i: INTEGER
    do
            from x := a until x >= b loop
                    Result := Result + f.item
                    i := i + 1; x := a + i * step
            end
    end
```

Объявление *f* указывает, что это функция с одним аргументом типа *REAL*, возвращающая значение типа REAL. Для вычисления значения функции в точке используем *f. item* ([*x*]). Ранее функция *item* для агентов уже была определена – она вызывает *call* и возвращает результат этого вызова. Эта функция ожидает кортеж в качестве аргумента, здесь ей передается манифестный кортеж.

Обратите внимание на вычисление заново *x* на каждом шаге вместо последовательного добавления шага. Это делается сознательно во избежание накопления ошибки.

Функция *integral* является частью класса *INTEGRATOR*, который описывает объекты, отвечающие за интегрирование математических функций. Если *your_integrator* – объект этого типа, то можно получить интеграл от функции *f* на интервале *a.. b* как значение

```
your_integrator. integral(agent f, a, b)
```

В классе *INTEGRATOR* следует объявить *step* как атрибут типа *REAL* со связанной сеттер-процедурой, что позволяет клиентам управлять точностью вычисления интеграла. Класс становится чем-то бóльшим, чем простой оберткой функции, он определяет абстракцию, имеющую практический смысл – интегрирование с управляемой точностью.

17.5. Открытые операнды

Иногда при использовании агентов необходимо больше гибкости, поскольку нужно помимо аргументов, передаваемых ассоциированному методу при вызове, задать некоторые дополнительные аргументы, но сделать это надо один раз и на все время определения агента. Мы будем говорить о «закрытых» и «открытых» аргументах.

Открытые аргументы

В качестве простого примера рассмотрим вариант последней схемы, где мы по-прежнему хотим вычислить интеграл от функции, но у функции есть дополнительные аргументы:

$$\int_a^b g\,(u, x, v)\,dx$$

Переменные u и v остаются константами во время интегрирования – только x, как и ранее, учитывается при интегрировании. Но значения переменных нужны для вычисления значения функции. Конечно, можно по-прежнему использовать предыдущее решение:

```
your_integrator.integral (agent g_extended, a, b)                    [13]
```

Необходимо определить функцию:

```
g_extended (x: REAL): REAL
        – Функция та же, что и g, с первым и третьим аргументами, равными u и v
    do
        Result := g (u, x, v)
    end
```

Предполагается, что u и v являются атрибутами класса *INTEGRATOR*. Это работает, но писать такие функции утомительно, и особенно неприятно, если u и v являются локальными переменными или формальными аргументами.

Функции, такие как *g_extended*, просто обертка, чья единственная цель состоит в заморозке некоторых аргументов функции, превращая ее в функцию только оставшихся аргументов. Делать это приходится часто, поэтому стоит ввести подходящее понятие. Тот же эффект, что и в **[13]**, можно получить без введения обертывающей функции, а используя выражение:

```
your_integrator.integral (agent g (u, ?, v), a, b)
```

Агентное выражение **agent** *g*(*u*, ?, *v*) обозначает функцию с одним аргументом, полученную из функции с тремя аргументами заморозкой первого и третьего аргумента значениями *u* и *v* соответственно. Истинным аргументом остается только аргумент во второй позиции, помеченный знаком вопроса.

Итак, при вызове агента задается список аргументов, соответствующий сигнатуре ассоциированной функции, но в этом списке любое число аргументов можно заменить знаком вопроса. Такие аргументы известны как открытые аргументы агента, а остальные, значения которых заданы при вызове, являются закрытыми аргументами агента. Агент рассматривает функцию только по отношению к открытым аргументам.

В частности, это означает, что наша первая нотация агента – **agent** *f* – является простым сокращением записи

```
agent f(?, ?, …)
```

Здесь, у функции открыты все аргументы. В этом случае проще использовать краткую форму: **agent** *f*.

Понятие открытых аргументов увеличивает многогранность агентов, избавляя от необходимости задания дополнительных функций – оберток, подобных *g_extended*. В качестве еще одного примера приведем вариацию схемы итерирования, включающей список целых. Рассмотрим программу:

```
increase_sum_by_power (?, n: INTEGER)
            -- Добавить к sum значение m в степени n.
    do sum := sum + m ^ n ensure added: sum = old sum +m ^ n end
```

Теперь *sum* типа *REAL*, так как значение этого типа возвращается после возведения в степень. После присваивания *sum*:= 0.0 можно получить сумму квадратов всех элементов списка il, вызвав:

```
il.do_all (agent increase_sum_by_power (?, 2)
```

Обобщая:

Определение: открытый и закрытый операнд

Операнд агента **закрыт**, если он специфицирован в момент *определения* агента. Операнд **открыт**, если он задается только внутри вызовов агента. В определении агента такой операнд помечается знаком вопроса – «?».

Терминологическое напоминание. «Определением агента» является выражение, его специфицирующее, такое как **agent** *f*(*a*, ?). Вызовом агента является оператор, вызывающий ассоциированный с агентом метод во время выполнения.

Открытые цели

В последнем определении введен новый термин – *операнд*. До сих пор мы говорили об открытых *аргументах*. Зачем же понадобилось новое понятие? Причина в том, что иногда необходимо не только сохранять открытым аргумент, но открытой должна быть и **цель** вызова.

Рассмотрим снова прежний пример с маршрутами и остановками:

```
your_route.do_all (agent print_stop_name)                    [14]
```

Вызов идентичен **[5]** за тем исключением, что в данном случае нет необходимости в процедуре *do_at_every_stop*, мы можем непосредственно вызывать *do_all*, так как в Traffic-класс *ROUTE* является фактически потомком *LINEAR*[*STOP*]. Пример предполагает процедуру *print_stop_name* с сигнатурой

```
print_stop_name (s: STOP)
```

Появляющийся в классе *C* **agent** *print_stop_name* имеет тип *PROCEDURE*[*C*, *TUPLE*[*STOP*]], соответствуя типу формального аргумента *do_all*.

Процедура *print_stop_name* на экземпляры *STOP* смотрит извне – она не принадлежит этому классу, но имеет аргумент типа *STOP*. Этот аргумент и будет тем самым открытым аргументом, так как запись **[14]** в реальности является сокращением для

```
your_route.do_all (agent print_stop_name (?) )               [15]
```

Всякий раз, когда в *do_all* вызывается метод *call* для агента, мы знаем, где это происходит: *action*. *call*[*item*] для каждого *item*, представляющего *STOP* в маршруте. Эффект от этого вызова тот же, что и для прямого вызова ассоциированного метода:

```
print_stop_name (your_route.item)                            [16]
```

Здесь подсветка в **[15]** и **[16]** показывает, что передается в качестве аргумента итерируемому действию. Схема итерации **[14]**, основанная на агентах, эквивалентна циклу, явно использующему *your_route*, инициализируя итерацию через *your_route.start*, продвигаясь по структуре *your_route.forth*, и выполняя вызов **[16]** на каждом шаге.

Подобно любому вызову в ОО-программировании, этот вызов имеет цель, но здесь цель задана неявно – текущий объект. Цель всегда можно сделать явной, записав вызов как **Current**.*print_stop_name* (*your_route*. *item*).

Предположим теперь, что мы рассматриваем не внешний, а внутренний метод класса *STOP*. Например, пусть в классе *STOP* существует метод *close*, отмечающий закрытые станции. Если мы хотим закрыть всю линию, то должны выполнить *close* для всех остановок. Но у метода *close* нет аргументов, он вызывается целью и применяется к ней; его типичный вызов:

```
some_stop.close
```

Так что действие, которое следует итерировать, теперь выглядит не как в **[16],** а так:

```
your_route.item.close
```

В данном случае целью метода, а не его аргументом, является то, что должно быть открытым в аргументе *do_all* и что должно заменить выражение **agent** *print_stop_name* (?) в **[15]** (или в краткой форме [14]).

Первое, что приходит в голову, – это написать нечто подобное ? *close*. Но это не работает, так как не задан тип цели, ведь многие классы могут иметь компонент *close*.

Мы должны задать тип цели и правильной формой для нашего примера является:

```
your_route.do_all (agent {STOP}.close)                                    [17]
```

Теперь вы видите, почему необходимо ввести общий термин – *операнд*: он покрывает все значения, необходимые для выполнения вызова – цель и аргументы.

Для открытых аргументов применяется простая запись с использованием знака «?», поскольку тип аргумента можно выяснить из известной сигнатуры метода, но можно применять и запись с явным указанием типа {*TYPE*}. Это может быть полезно, если указывается тип, отличный от типа сигнатуры, но, естественно, согласованный с ним.

Все комбинации открытых и закрытых операндов являются правильными. Предположим, что *f* и *g* – методы класса *C*; *f* имеет аргументы, а *g* – без аргументов. Тогда возможно:

- все закрыто: **agent** *f*(*x*, *y*, *z*), **agent** *g*;
- цель закрыта, все аргументы, если есть, открыты: **agent** *f*(*?*, ?, ?), **agent** *g* (возможна краткая форма записи в этом случае - **agent** *f*);
- цель закрыта, некоторые аргументы открыты, некоторые – закрыты: **agent** *f*(*?*, *y*, *?*);
- цель открыта, некоторые аргументы открыты, некоторые – закрыты: **agent** {*C*}.*f*(*?*, *y*, *?*);
- все открыто: **agent** {*C*}.*f*.

Эти механизмы позволяют нам иметь единое множество итераторов – *do_all*, *do_if* и другие в *LINEAR* и его потомках. Без этого пришлось бы иметь два множества: одно для целей, другое – для аргументов.

17.6. Лямбда-исчисление

Мы уже познакомились с основами агентов и даже некоторыми деталями. Надеюсь, вы ощутили мощь этого механизма и возможные области его применения.

Фактически возможен следующий уровень гибкости. Прежде чем его освоить, следует рассмотреть математические идеи, лежащие в основе. Лямбда-исчисление даст нам более глубокое понимание агентов, в частности, концепции открытых и закрытых операндов.

Эта замечательная теория разработана в 1930 году еще до появления компьютеров. Через 30 лет обнаружилось, что она дает ясную и прочную основу многих концепций языков программирования. Это оживило интерес к лямбда-исчислению, и оно стало плодотворной областью исследований.

Лямбда-исчисление дает нам теорию понятия функции. Оно занимается основами понятия, не изучая специальные виды функций, таких как функции тригонометрии или функции вещественных переменных. Здесь изучается сама идея функции как механизма, принимающего аргументы и вырабатывающего результат. Это математическое понятие, но оно весьма схоже с понятие программы в компьютерных науках. Теория позволяет нам лучше понять, что является областью определения переменной, какова роль аргументов, а также возможность рассматривать программу как объект, отвечая целям этой главы – преобразовать программы в агенты.

Операции над функциями

Основная идея проста: нотация и правила трансформации позволяют нам обращаться с функциями так же, как и с другими математическим объектами.

Для заданных двух чисел a и b можно образовывать различные комбинации, например: $a + b$ или $\sin(a) + \cos(b)$, используя функции с хорошо определенной сигнатурой.

```
sin: REAL → REAL                - Смысл: Для любого аргумента типа REAL функция sin
                                - вырабатывает результат типа REAL
"+": [REAL × REAL] → REAL
                                - × декартово произведение, квадратные скобки
                                - применяются для группировки аргументов
```

Можно ли «играть» в подобные игры не с числами, а с функциями? Даже в элементарной операции мы встречаемся с операциями над функциями. Если *f* и *g* – это функции с подходящими сигнатурами, то можно задать их композицию, записываемую как $g \circ f$ или *f; g* (нотация, которую мы будем использовать, поскольку она явно указывает порядок выполнения). Результатом композиции является функция $h(x)$, такая, что $h(x) = g(f(x))$ для любого применимого аргумента *x*. Композиция является такой же операцией над функциями, как «+» над числами.

Лямбда-исчисление позволит нам определить над функциями много операций.

Мы можем продолжить восхождение по лестнице абстракций. Композиция функций является функцией, поэтому и к ней применима композиция. Пусть *X*, *Y*, *Z* – некоторые множества и заданы сигнатуры функций *f* и *g*:

```
f : X → Y
g: Y → Z
```

Композиция этих функций, названная выше функцией *h*, имеет сигнатуру $X \to Z$. Определим теперь композицию «; » как функцию, которой передаются два аргумента *f* и *g* и которая вырабатывает результат *h*. Эта функция имеет сигнатуру:

```
";": [[X → Y ] × [Y → Z ]] × [X → Z]                                    [18]
```

Мы можем продолжить определение функций, которые оперируют функциями, которые, в свою очередь, оперируют функциями, и так далее. Лямбда-исчисление дает нам словарь и правила – теорию – для работы с такими функциями на произвольном уровне.

Лямбда-выражения

Прежде всего, нам необходима простая нотация для определения функций. Будем предполагать, что у нас есть базис из функций, таких как «+», над целыми и вещественными. Это предположение делается для простоты, так как лямбда-исчисление может быть определено без ссылок на существующие математические теории. Символ $\triangleq$ будет, как обычно, означать «по определению». Определим функцию *square* с сигнатурой

```
square : REAL → REAL
```

Функция в качестве результата возвращает квадрат числа. Ее определение можно задать соответствующим **лямбда-выражением:**

```
square ≜     λ x: REAL | x * x                                          [19]
```

Правая часть определения является лямбда-выражением, запись которого однозначно позволяет установить, что для любого *x* типа *REAL* результатом функции будет *x* * *x*.

Символ λ (лямбда) – дело соглашения, но он дал имя всему подходу. В математической литературе сигнатуру от определения функции отделяет символ точки, но в ОО-программировании точка играет другую важную роль, поэтому вместо точки применяется символ « | » вертикальной черты.

Лямбда-определение функции напоминает ее определение в программировании

```
square (x: REAL): REAL                                        [20]
            -- x в квадрате.
  do
            Result := x * x
  end
```

Как правило, математическая нотация всегда компактнее программистской. В этой нотации переменная, следующая за λ, называется **связанной переменной** лямбда-выражения, она подобна формальному аргументу метода.

Подобно имени формального аргумента, имя связанной переменной не влияет на смысл выражения и может быть любым. Так что следующее определение задает ту же самую функцию, вычисляющую квадрат числа:

```
λ y: REAL | y * y
```

Это наблюдение будет формализовано ниже, используя понятие *альфа-преобразования*.

Лямбда-выражение может иметь более одной связанной переменной, требуется лишь, чтобы у них были разные имена:

```
λ x, y: INTEGER | x + y              -- Функция сложения
λ x: NATURAL, z : REAL | z^x         -- Нотация для случая разных типов
```

Чем хорошо лямбда-выражение? На первый взгляд, вместо него можно было бы использовать уже знакомую запись, например, для функции *square*:

```
∀ x : REAL | square (x) = x * x                                [21]
```

Разница в том, что **[21]** определяет **свойства** функции, в то время как **[19]** определяет **функцию** – математический объект со всеми правами, аналогично тому, как можно определить константу π, задав ее значение.

Одним из непосредственных преимуществ является возможность определения функций высших порядков, таких как композиция, сигнатура которой задана в **[18]:**

```
";" ≜ λ f : X → Y, g : Y → Z | g (f (x))
```

Предполагается, что множества *X*, *Y*, *Z* известны. Так как они произвольны, мы можем ввести механизм универсальности в лямбда-выражение, как для классов, превращая имена множеств в формальные родовые параметры. В нашем кратком обзоре нет необходимости в такой нотации.

В этом примере исходное множество в сигнатуре является декартовым произведением $[X \to Y] * [Y \to Z]$; соответственно, лямбда-выражение имеет две связанные переменные – f и g.

До сих пор каждому определению функции лямбда-выражением предшествовало задание сигнатуры функции, а каждая связанная переменная сопровождалась указанием ее типа. В принципе, возможно нетипизированное лямбда-выражение, но мы будем продолжать использовать только типизированные, аналогично тому, как мы используем в программировании типизированные языки, такие как Eiffel, повышая читабельность и избегая ошибок.

Почувствуй методологию

Объявление сигнатуры

Объявление сигнатуры функции должно предшествовать определению функции лямбда-выражением.

Если сигнатура появляется непосредственно перед определением, то в определении можно опускать связанные переменные, как в этом примере:

```
";": [[X → Y] * [Y → Z]] * [X → Z]
";" ≜ λ f,g| g (f (x))- Нет необходимости в объявлении f и g.
```

Карринг

В качестве примера функции высшего порядка, которую можно описать лямбда-выражением, рассмотрим *карринг*.

Карринг назван в честь американского математика Карри Брукса Хаскелла – одного из основателей теории, известной как комбинаторная логика, частью которой является лямбда-исчисление. Карринг, без потери общности, позволяет рассматривать функции, имеющие только один аргумент. Вначале соглашение, связанное с нотацией:

Почувствуй нотацию

Квадратные и круглые скобки

В обычной математической нотации круглые скобки служат как для группирования, так и для записи функций. В примере $f(a * (b + c))$ внутренние скобки используются для группирования, внешние – для записи функции. Это приводило бы к непониманию при обсуждении операций над функциями.

При обсуждении лямбда-исчисления круглые скобки будут применяться только при задании функции, а для группировки используются квадратные скобки. Так, запись:

$[f ; g](a * [b + c])$

означает применение композиции функций f и g к аргументу, заданному выражением – произведением a и суммы $b + c$.

Часто приходится иметь дело с бинарными функциями, имеющими два аргумента, такими как композиция «;» « или сложение «+». Рассмотрим такую функцию:

```
f : [ X × Y ] → Z
```

для заданных X, Y, Z. Зная f, определим функцию f' с сигнатурой:

```
f' : X → [Y → Z]
```

как

```
f' ≜ λ x : X | [λ y:Y|f(x,y)]                                  [22]
```

Что это означает? В отличие от f функция f' принимает только один аргумент типа X, а также в отличие от f не возвращает результат типа Z. Вместо этого она для любого x в качестве результата возвращает функцию, заданную в **[22]**. Давайте назовем эту функцию g. Эта функция от одного аргумента y типа Y возвращает результат типа Z. Результат $g(x)$ – тот же, что и $f(x, y)$, как если бы сразу применили пару аргументов к функции f.

Функцию f' называют карризованной версией функции f. Карринг двухаргументной функции означает преобразование ее в одноаргументную функцию, связанную с оригиналом соотношением **[22]**. Говорят также, что карринг означает специализацию функции по первому аргументу. Специализация, связывая первый аргумент, оставляет свободным только второй аргумент, что и преобразует функцию в одноаргументную.

Если функция *add* – сложение целых, определение которой можно задать лямбда-выражением $add \triangleq \lambda\ x, y : INTEGER \,|\, x + y$, то *curry*(*add*) является функцией:

```
add' ≜ λ x : INTEGER | [λ y : INTEGER | x + y]
```

Так что $add'(1)$ - $\lambda y\text{: } INTEGER \,|\, 1+y$ – это функция « плюс 1», добавляющая 1 к заданному числу.

Соответствие между двухаргументной функцией f и ее карризованной версией f' взаимно однозначно. Неформально при проведении карринга никакая информация не теряется, так как эффект второго аргумента остается встроенным в аргумент результирующей функции f'.

Будет интересно – и послужит примером выразительной силы лямбда-нотации – явно установить соответствие между *f и f'*, рассматривая карринг как функцию *curry*, определяемую лямбда-выражением. Для заданных X, Y, Z ее сигнатура:

```
curry: [[X × Y ] → Z ] → [X → [Y → Z ]]
```

Ее значение

```
curry ≜ λ f : [X × Y ] → Z | [λ x : X | [λ y : Y | f (x, y)]]
```

Аналогично определите обратное соответствие, создающее f по функции f'.

Обобщение карринга

Во всех примерах карринг применялся к двухаргументной функции по первому аргументу. Достаточно просто обобщить концепцию: карринг можно применять к любой функции из n ($n \geqslant 1$) аргументов для любого выбора m аргументов ($1 \leqslant m \leqslant n$), задав значения для выбранного набора аргументов. Это превращает исходную функцию в функцию с $n - m$ аргументами, представляя специализированную версию исходной функции, также известной как **частичное вычисление**. Если $m = n$, то получаем константную функцию.

Карринг на практике

Как пример того, что карринг представляет на практике, рассмотрим разницу между *компиляцией* и *интерпретацией*.

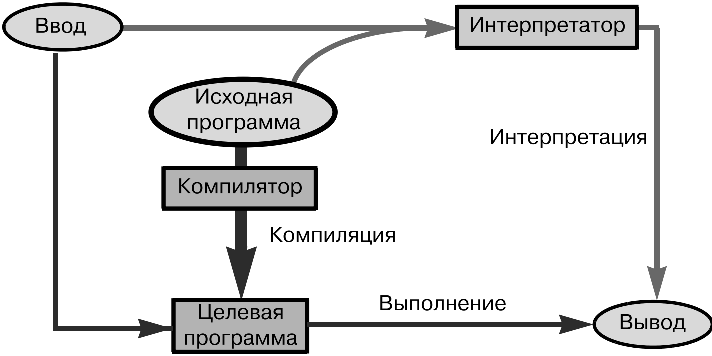


Рис. 17.7. Интерпретация и компиляция

Интерпретатор с абстрактной точки зрения можно рассматривать как функцию с сигнатурой:

```
interpreter : Program × Input → Output
```

Здесь *Program* — множество всех корректных программ, *Input* и *Output* — множества возможных входов и выходов (это упрощенная, но достаточно корректная точка зрения на программы). Компилятор создает из исходной программы машинный код, который на компьютере уже без дополнительных усилий может быть выполнен на некотором входе:

```
Machine_program ≜ Input → Output
```

Абстрактно работу компилятора можно рассматривать как функцию с сигнатурой:

```
compiler : Program → [Input → Output]
```

Когда у нас есть два механизма выполнения для одного и того же языка программирования, важно, чтобы они реализовали одну и ту же семантику.

Это основа работы в EiffelStudio, где постоянно приходится переключаться от полностью скомпилированной, полностью оптимизированной — финальной формы компиляции к быстрой возрастающей перекомпиляции — «технологии тающего льда», где главным образом применяется интерпретация. Конечно, при поставке конечного продукта выполняется финальная компиляция, но результаты работы совпадают с версией «тающего льда».

Требование согласованной работы компилятора и интерпретатора точно и элегантно выразимо с использованием карринга:

```
compiler = curry (interpreter)
```

ОО-стиль программирования отвечает духу карринга. ОО-вычисления инициируются вызовом:

```
x. f (args)
```

Фиксирован объект – цель вызова, который и вызывает операцию. Для неквалифицированных вызовов – *f*(*args*) неявно заданной целью является объект **Current** и вызов можно записать в виде **Current**. *f*(*args*).

В ОО-программировании никогда не говорят «Примени эту операцию к тем объектам», что характерно для стиля, отличного от ОО, применяющего вызовы в форме *f*(*x*, *arg1, arg2,* ...) со всеми операндами на равной ноге. ОО-программисты говорят «Этот объект применяет операцию, и, если нужны некоторые аргументы, то вот они».

Конечно, все, что выразимо в одном стиле, можно выразить и в другом. Но привычки ОО-стиля, влияющие на структуру программы, глубоки. Напомним только о двух важных понятиях – классе, играющем роль модуля и типа данных, и о наследовании.

Все это прячется в концепциях этого раздела. ОО-программирование является карринг-программированием.

Исчисление

В лямбда-исчислении до сих пор мы познакомились только с лямбда-выражением – нотация полезна для понимания, но все же это далеко не исчисление. Основываясь на нотации, лямбда-исчисление дает замечательную теорию функций и операций над ними. Нам удастся познакомиться только с основными идеями, поскольку детальное рассмотрение не является предметом данной книги.

Лямбда-исчисление дает модель общего понятия – *вычисление* введением двух основных операций над лямбда-выражениями – альфа-преобразования и бета-редукции (кратко записываемых также как α- и β-).

Для определения этих понятий нам необходимо отличать два вида вхождений переменных в лямбда-выражение – связанные и свободные вхождения.

Как вы помните, мы говорили, что *x*, *y*, ... являются *связанными переменными* в лямбда-выражении $\lambda\, x{:}\ X,\ y{:}\ Y \ldots|$ e. Тогда нетрудно определить понятие связанного **вхождения**. Вхождение переменной *a* в лямбда-выражение является **связанным**, если:

- *a* – одна из связанных переменных;
- вхождение является (рекурсивно) связанным вхождением *a* в *e*.

Понятие немедленно обобщается на выражение *exp*, не являющееся лямбда-выражением: вхождение является связанным в *exp*, если оно является связанным в одном из его лямбда-подвыражений. Например:

```
[f ; g] (λ a : INTEGER | a + f (a, b))
```

Здесь вхождение *a* является связанным, но это не так для *f, g* и *b* в данном примере – это **свободные** вхождения.

В следующем примере:

$$\lambda\, x : INTEGER \mid [\lambda\, y : INTEGER \mid x + y + z\,]$$

вхождения *x* и *y* связаны, но вхождение *z* свободно. Неформально это означает, что *x* и *y* являются локальными переменными выражениями, в то время как переменная *z* должна быть

определена вне выражения. Это в точности соответствует тому, что мы имеем в программировании:

```
f (x, y: INTEGER): INTEGER do Result := x + y + z end
```

Здесь *x* и *y* — формальные аргументы, которые означают удобные имена, используемые при определении функции; любые другие имена работали бы точно так же при условии отсутствия конфликта с другими именами. Переменная *z* имеет другой статус и должна быть определена в контексте. На практике она должна быть компонентом класса — запросом (атрибутом или функцией без аргументов).

> Будем говорить, что *x* «входит связно» в выражение *exp*, если имеет по меньшей мере одно связанное вхождение в *exp*, и что «входит свободно», если имеет по меньшей мере одно свободное вхождение в *exp*, во втором случае *x* – свободная переменная в *exp*.

Другим базисным понятием является подстановка.

Определение: подстановка переменной

> Пусть *exp* – выражение, *x* – переменная, а *e* – другое выражение. Тогда
>
> *exp* [*x*:= *e*]
>
> обозначает выражение, полученное из *exp* путем подстановки (замены) каждого *свободного* вхождения *x* на выражение *e*.

Например, если *exp* является:

$$\lambda z : INTEGER \mid x + y + z \times x$$

и *e* — это *sin*(*x*), то *exp* [*x*:= *e*] — это выражение $\lambda z: INTEGER \mid sin(x) + y + z * sin(x)$

Как показано в примере, выражение может содержать несколько вхождений переменной. Заметьте, подстановка выполняется только для свободных вхождений. Если *exp*:

$$\lambda x, z : INTEGER \mid x + y + z \times x$$

Данное выражение отличается от предыдущего тем, что теперь *x* является связанной переменной. После аналогичной подстановки *exp* [*x*:= *e*] выражение не изменится, поскольку нет свободных вхождений *x*. Если *exp*:

$$\lambda y : INTEGER \mid f(x, [\lambda x : INTEGER \mid x + y])$$

то подстановка заменит только первое вхождение *x*, являющееся свободным, но не связанную переменную *x* лямбда-выражения. Альфа-преобразование прояснит ситуацию.

Но начнем рассмотрения с **бета-редукции** — центрального правила, охватывающего суть лямбда-нотации. Бета-редукция позволяет нам избавиться от связанной переменной (а следовательно, если нет других переменных, то и от λ) преобразуя

```
[λ x : X | exp] (e)
```

в

```
exp [x := e]
```

Это справедливо при условии, что **нет свободной переменной выражения *e*, связно входящей в *exp***. Это четко выражает понятие применения функции к фактическим аргументам, так как запись $\lambda\, x : X \mid exp$ интуитивно означает, что *exp* рассматривается как функция с аргументом *x* и что подстановка выражения *e* вместо *x* означает замену всех свободных вхождений *x*. Вместо слов «бета-редукция трансформирует *e* в *f*» будем использовать обозначение $e \xrightarrow[\beta]{} f$:

```
[λ x : X | x + y] (z)  →β  z + y
[λ x : X | x + y] (y)  →β  y + y
[λ x : X | x + y] (x)  →β  x + y
[λ x : X | z + y] (e)  →β  z + y
```

В последнем примере связанная переменная фактически не используется в *exp*; в этом случае можно рассматривать лямбда-выражение как константную функцию от *x*.

Как показывают второй и третий примеры, бета-редукция возможна и в том случае, когда *e* использует переменные, встречающиеся в *exp*, лишь бы они были не связанными. Даже третий пример не нарушает ограничение, поскольку в выражении *exp* ($x + y$) переменная *x* не связана – она связана в охватывающем лямбда-выражении, но не в *exp*. Ограничение имеет место, предотвращая бета-редукцию, только в случаях, подобных данному:

```
[λ x : X | [λ y : Y | x + y]] (y)                                    [23]
```

Здесь редукция приводила бы к выражению $\lambda\, y: Y \mid y + y$, что некорректно, так как порождало бы новые вхождения связанной переменной, не соответствующие неформальному пониманию лямбда-выражения.

Значит ли это, что в подобных случаях бета-редукция невозможна по той причине, что нам не повезло с именем? Это было бы огорчительно, так как имена связанных переменных произвольны и их можно выбирать, не меняя общего смысла. Если мы заменим [23] на:

```
[λ x : X | [λ z: Y | x + z]] (y)
```

то бета-редукция становится возможной, давая результат $\lambda\, z: Y \mid y + z$.

> В программировании мы делаем то же самое, когда выбираем новое имя для формального аргумента метода, если оно конфликтует с именем атрибута класса.

Для узаконивания таких безвредных изменений связанных переменных нам необходимо второе правило – **альфа-преобразование**. Для заданной переменной *y* альфа-преобразование трансформирует лямбда-выражение

$\lambda\, x : X, \ldots \mid exp$

в котором y не имеет ни свободных, ни связанных вхождений в выражение:

$\lambda\, y : X, \ldots \mid exp\ [x := y]$

Условие, налагаемое на y, защищает от замены x на y в обоих ниже приведенных случаях:

$\lambda x : X \mid x + y$ **[24]**
$\lambda\, y : X \mid x + y$ **[25]**

При замене результирующее выражение $\lambda\, y$: $Y \mid y + y$ потеряло бы семантику, которая подразумевается в исходном выражении.

- **[24]** представляет функцию одного аргумента, возвращающую значение y, к которому добавлено значение, переданное функции в качестве аргумента. Для этой функции y – свободная переменная, определенная в контексте выражения (например, в охватывающем выражении). Если же в данном случае подстановка была бы допустимой, то полученное выражение задавало бы функцию, удваивающую значение переданного ей аргумента. Две функции полностью различны!
- В **[25]** y связано, но тогда альфа-преобразование сливало бы y со свободной переменной x.

Последнее наблюдение показывает, что требование на y избыточно, так как в **[25]** предварительно y можно переименовать.

Альфа-преобразование и бета-редукция дают основу для полностью проработанной теории вычислений, описывающей любые вычисления как последовательность таких преобразований (возможно, сложную) лямбда-выражений. Фундаментальное свойство согласованности этой теории выражает теорема Черча – Россера.

Теорема гласит, что если из данного лямбда-выражения *exp* две различные последовательности трансформаций приводят к различным выражениям *exp1* и *exp2*, то существуют две другие последовательности трансформаций, которые приводят оба эти выражения к единому выражению *f*. Это означает, что, если возможны некоторые трансформации для любого частного выражения, то не имеет значения, с какого преобразования начинать, поскольку в конечном итоге придем к одной и той же канонической форме.

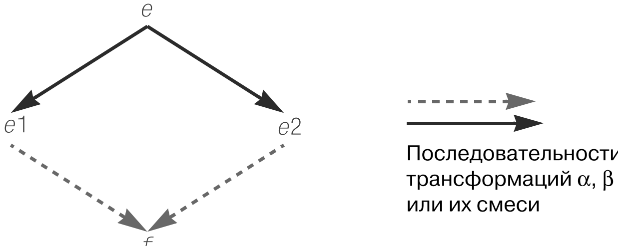


Рис. 17.8. Свойство Черча – Россера

Лямбда-исчисление и агенты

Я надеюсь, что, прочитав о лямбда-исчислении, вы начали искать связь с концепциями предыдущих разделов.

Методы, как мы знаем из предыдущих глав, представляли для нас структурные конструкции нашей программы, но они не участвовали в играх при выполнении программы, как это делают ссылки, базисные объекты, такие как целые, и более сложные объекты. Подобно математическим функциям в отсутствие каркаса, подобного лямбда-исчислению, методы оставались – в соответствии с метафорой, широко используемой в литературе по ОО-языкам программирования, – «гражданами второго сорта».

Аналогично тому, как лямбда-исчисление превращает функции в «первосортных граждан математического мира», так и агенты причесывают методы, одевая их в объектные одежды, наделяя их привилегиями значений, готовых для программного выполнения.

Даже в отсутствие механизма агентов методы являются формой лямбда-выражений, и вызов метода является формой бета-редукции. Но эта редукция должна планироваться статически, через вызовы, такие как $f(x, y)$ с явно заданным методом f. ОО-программирование вводит первый элемент динамизма благодаря динамическому связыванию, разрешая f иметь несколько вариантов, выбор между которыми делается при каждом вызове $a.f(x, y)$ на основе типа объекта, присоединенного к a. Такой динамический механизм позволяет нам представить ряд примеров в виде образца «много маленьких оберток» (с цитированными ограничениями), но предлагаемый выбор ограничен множеством построенных вариантов. С агентами бета-редукция становится полностью динамической операцией, вызов *a.call*([x, y]) не требует от нас какого-либо знания о методе, который ассоциирован с агентом, за исключением знания сигнатуры.

Концепция лямбда-исчисления помогает нам понять природу «открытых» и «закрытых» операндов. Они у агентов соответствуют связанным и свободным переменным лямбда-выражений.

- В $e \triangleq \lambda x: INTEGER \mid x + y$ связанная переменная x представляет аргумент, который будет задан во время бета-редукции, свободная переменная приходит из окружения, типично из охватывающего выражения.
- В **agent** *f(?, y)* открытый аргумент будет задан во время вызова, закрытый аргумент обеспечивается при определении.

Вы могли обратить внимание на то, что задание закрытых аргументов, по сути, означает выполнение карринга (в его общей форме – фиксация любых *m* из n аргументов). Рассмотрим различные динамические формы, которые можно получить из метода.

- **agent**{*C*}. *f* – динамическая версия *f*, в полной мере соответствующая сигнатуре оригинала.
- Другой крайний вариант – **agent** $f(x, y)$ и **agent** $t.f(x, y)$, где все операнды закрыты, полностью соответствует карринг-версии, так что можно вызывать (если *a* агент) *a.call*([]) без всяких аргументов. Это, кстати, напоминает нам разницу между математикой и программированием: математическая функция при выполнении карринга по всем аргументам превращается в константу, в то же время успешные вызовы *a.call*([]) могут давать разные результаты, поскольку даже если *a* не изменяется, могут изменяться окружающие объекты.
- Посередине вариант с агентом, имеющим как открытые, так и закрытые операнды, такие как **agent** $a.f(?, x, y)$, подобный функции с каррингом на закрытых операндах.

В одном отношении агенты, которые мы видели до сих пор, менее общие, чем лямбда-выражения. Чтобы использовать **agent** $a.f(?, x, y)$ или любой другой вариант, мы должны предположить, что функция f построена. Такое предположение для лямбда-выражения λx ...| *exp* означало бы ограничение *exp* формой *f*(*args*). Теперь покажем, как для агентов можно снять это ограничение, позволяя агентам иметь произвольную форму, как это происходит для *exp* в лямбда-выражении.

17.7. Манифестные агенты, непосредственно определяющие функцию

Изучение лямбда-исчисления позволяет обобщить базисный механизм агентов, придавая дополнительную гибкость.

Агенты, изучаемые до сих пор, связывались с существующими методами класса. Но иногда хочется иметь агента и не иметь метода в классе. Представьте себе, что нужно выполнить с использованием агента некоторое простое вычисление. Кажется излишним во всех случаях создавать в классе метод, который может и не отражать свойства класса. Метод может понадобиться только в одном месте, и нет смысла перегружать им класс[1]. Манифестные агенты (inline) позволяют определить агент, не тревожа никакой класс.

Такая необходимость часто возникает при написании контрактов – всех видов предусловий, постусловий, инвариантов класса. Например, инвариант класса может специфицировать, что все элементы некоторого массива целых являются положительными. Мы уже знаем, как это установить, благодаря классу *INTEGER_INTERVAL* и оператору |..|, рассматриваемому в параграфе 17.3 этой главы. Мы видели, как установить требуемое условие, эквивалент выражения _ *s: a.lower..a.upper* | *a*[*i*] > 0 в исчислении предикатов:

```
(a.lower |..| a.upper). for_all (agent is_positive)                    [26]
```

Чтобы сделать это условие выполнимым, следует написать небольшую функцию для данного случая.

```
is_positive (n: INTEGER): BOOLEAN                                      [27]
        – Больше ли n нуля?
    do
        Result := (n > 0)
    ensure
        definition: Result = (n > 0)
    end
```

Немного досадно. Не так уж много времени требуется для написания кода. Никогда не жалейте потратить время на нажатие клавиш для получения релевантного результата. Но, представьте, эта функция нужна только для того, чтобы выразить данное свойство **[26]** в инварианте класса. Тогда вы загромоздите класс компонентом, который не представляет соответствующий абстракцию данных класса. Это становится особенно неприятно, если таких свойств много, как происходит, когда мы даем ясное и точное описание контрактов класса. Это правда, что эти компоненты не требуется экспортировать, но они в любом случае становятся частью класса. Было бы лучше выразить релевантные свойства точно в том месте, где возникает в них необходимость, но без видимости их вне этого контекста.

Манифестные агенты в полной мере отвечают поставленной задаче. Манифестный агент, как говорит его имя, задает объявление метода в стиле, подобном объявлению метода, но

[1] Ситуация похожа на ситуацию с константами. Если константа используется многократно, то следует создавать именованную константу. Для констант однократного употребления иногда полезнее и понятнее использовать в точке ее применения манифестную константу. Аналогично – манифестный агент может непосредственно в точке вызова определить вызываемую функцию.

ничего более не требуется, никакие методы класса не появляются. Синтаксис непосредственно выводится из переписи нашего последнего примера. Мы, по сути, сливаем **[27]** в **[26]**, получая в результате:

```
(a.lower |..| a.upper). for_all
     (agent (n: INTEGER): BOOLEAN                                        [28]
                              — Больше ли n нуля?
             do
                     Result := (n > 0)
             ensure
                     definition: Result = (n > 0)
             end)
```

Начиная со второй строчки текст совпадает с **[27],** с тем исключением, что исчезает ненужное теперь имя метода (агент манифестный).

Манифестный агент характеризуется следующим свойством: он задает **анонимный метод**.

Синтаксис манифестного агента, как показано в примере, совпадает с синтаксисом объявления метода с заменой имени метода на ключевое слово **agent**. Разрешается включать все компоненты, применимые к методу, такие как предусловия и постусловия, вводить собственные локальные переменные, чьи имена должны отличаться от имен методов класса и локальных переменных охватывающих методов. Это не соответствует соглашению, принятому для лямбда-выражений, где внутреннее связывание сильнее внешнего, но помогает избежать недоразумений. В конце концов, имена не являются критическим ресурсом, выбор их должен доставлять удовольствие, а формально переименование означает применение альфа-преобразования.

Даже когда нет конфликта с именами локальных переменных охватывающих методов, эти переменные нельзя непосредственно использовать в агенте. Если такая редкая необходимость возникает, то следует применить передачу переменных через аргументы агента.

Манифестные агенты дополняют механизмы агентов, обеспечивая поддержку выразительности наших ОО-программ. В следующей главе подробно рассмотрим главное приложение этого механизма, которое позволяет построить элегантное решение проблемы «наблюдения», кратко охарактеризованной в этой главе.

17.8. Конструкции в других языках

В начале этой главы обсуждались ситуации, которые требуют использования объектов, представляющих обертку вычислений. Агенты являются эффективным инструментом в подобных случаях.

Не все языки программирования, однако, обладают такие конструкциями. Фактически среди языков, применяемых в индустрии, только Eiffel, Smalltalk и C# имеют похожие средства (существенно отличаясь в деталях). Представляет интерес вкратце рассмотреть, какие же решения являются доступными в зависимости от языка, который, возможно, вы используете.

Применяются четыре основных подхода:

- механизм, поддерживающий лямбда-выражения, такой как агенты;
- методы, как аргументы других методов;
- указатели функций;
- образец «много маленьких оберток».

Механизмы, подобные агентам

Язык C# вводит понятие делегатов, предназначенных для тех же целей, что и агенты.

Если не принимать во внимание дух и нотацию, можно сказать, что главная разница между делегатами C# и агентами Eiffel в том, что цель делегата не может быть открытой. Выражение **agent** {*STOP*}.*close* не имеет прямого эквивалента в C#. В приложении, посвященному языку C#, о делегатах говорится подробнее.

Язык Smalltalk вводит понятие блока (block) – сегмента кода, который может передаваться как объект. Заметьте, что Smalltalk является нетипизированным языком, поэтому здесь нет способа проверить во время компиляции, что передаваемый блок будет использоваться с подходящими аргументами, – любые несоответствия приводят к появлению ошибок в период выполнения.

Функциональные языки типично поддерживают возможность рассматривать функции как данные. Это пришло еще от языка Lisp, где выражение в форме

```
(defun f (x y) (“expression involving x and y”))
```

определяет f как функцию двух аргументов. Тогда можно использовать f как аргумент другой функции, например:

```
(curry f)
```

Сама функция curry может быть определена в Lisp. Язык был определен на базе бестипового лямбда-исчисления, так что нет ничего удивительного, что многое из того, что мы видели в этой главе, без труда выразимо в Lisp. Все это применимо и для более современных функциональных языков, таких как Haskell и ML. Следует отметить два момента.

- Функциональные языки изначально не являются ОО-языками. Некоторые из них добавляют конструкции, такие как класс и наследование, но не все конструкции, которые хотелось бы сохранить, применимы в функциональном окружении.
- В то время как некоторые функциональные языки являются статически типизированными, другие таковыми не являются. Получение преимущества от статической типизации зависит от выбранного варианта языка.

Термин «замыкание» часто используется в функциональных языках для обозначения выражений, которые могут передаваться как данные, даже если они могут нуждаться в доступе к глобальным переменным.

Программы как аргументы

Некоторые языки программирования позволяют передавать методы как аргументы другим методам с примерно таким синтаксисом:

```
integral (f:function(x: REAL):REAL ; a, b: REAL): REAL
```

В этом случае методу *integral*, вычисляющему интеграл, можно передать в качестве фактического аргумента метод с соответствующей сигнатурой, вычисляющий подынтегральную

функцию. Язык должен обеспечить подходящую нотацию для вызова соответствующего метода из кода метода, такого как *integral*.

В сравнении с агентами или замыканиями такое решение имеет ограничения.

- «Метод» является аргументом специального типа, который не соответствует в полной мере системе типов языка.
- Обычно информация о методе не представляет хорошо определенное значение, как в случае агента или замыкания, и, следовательно, метод не может быть присвоен переменной (для которой, учитывая предыдущий пункт, было бы трудно задать соответствующий тип) – он может быть только использован как аргумент.
- Из-за отсутствия подходящей типизации аргументов невозможно или, по крайней мере, не просто перейти на следующий уровень абстракции и определить функции, такие как композиция или карринг.
- Все, что можно делать с аргументом, представляющим метод, – это вызвать его. В противоположность этому агенты являются полноправными объектами, чьи компоненты обеспечивают информацию об ассоциированном методе.
- Могут возникать проблемы, когда методам нужен доступ к глобальным переменным. В первую очередь проблемы появляются у разработчиков компиляторов, но могут касаться и программистов.
- Подход не согласуется в полной мере с ОО-схемой, так как использует данные, не являющиеся объектами.

Однако этот подход удовлетворяет многим основным потребностям, он успешно применяется в не ОО-языках, начиная с Фортрана и продолжаясь в Паскале и его последователях.

Указатели функций

Компьютеры, как вы знаете, используют память для хранения не только объектов, но и программ. Во время выполнения у каждой конкретной программы свой конкретный адрес памяти, где она хранится. Это делает возможным передать управление коду по адресу его расположения: если есть способ для программы обозначить свой адрес и существует механизм, допускающий команду типа «выполнить метод по адресу *addr*, а затем вернуться и продолжить», то можно рассматривать адреса методов как данные, благодаря которым вызываются соответствующие методы. На машинном уровне эти приемы и обеспечивают нужные потребности.

- Когда вы используете метод как аргумент другого метода, компилятор, фактически, будет передавать адрес метода.
- Объект, представляющий агента, в одном из своих полей (недоступных клиенту по понятным причинам) будет хранить адрес ассоциированного метода.
- Динамическое связывание, необходимое для образца «много маленьких оберток», предполагает способность вызывать метод по его адресу, который хранится в некоторой структуре данных, представляющей свойства типа. Таблица методов, которая описывалась в этой главе при рассмотрении приемов реализации наследования, является примером такой структуры.

Все эти приемы важны для компилятора в процессе генерирования кода, а не для прикладного программиста, когда он пишет свою программу. Компилятор скрывает адреса методов под одним или несколькими слоями абстракции, позволяя программисту думать в терминах более высокого уровня – методах, объектах, агентах.

Языки C и C++ позволяют передавать имя функции (процедуры рассматриваются как функции, возвращающие значение void) как фактический аргумент или присваивать его пе-

ременной. Тогда, если *f* является соответствующим формальным аргументом или переменной, можно вызвать функцию следующим образом:

```
(*f ) (args)
```

Когда объявляется формальный аргумент, представляющий функцию, можно специфицировать его сигнатуру, известную как прототип, так что фактический аргумент, не соответствующий сигнатуре, должен быть отвергнут во время компиляции. Однако не обязательно задавать сигнатуру. Можно обойтись без этого, потеряв возможность получения предупреждений во время компиляции. Принимая это и рассматривая имя функции как ее адрес, получаем ту же гибкость, что и при программировании на языке ассемблера, теряя преимущества статической проверки типов.

«Много маленьких оберток» и вложенные классы

Если язык программирования не поддерживает ни одну из упомянутых техник, но является ОО-языком – с классами, наследованием, полиморфизмом и динамическим связыванием, – то можно использовать образец «много маленьких оберток», изученный в начале этой главы.

Его главный недостаток – необходимость писать много маленьких классов, часто включающих только один метод.

Язык Java, не имеющий механизма, подобного агентам, и не позволяющий передавать методы в качестве аргументов, смягчает проблему, позволяя программисту объявлять класс, локальный по отношению к другому классу, что также известно как вложенный класс. Тогда можно использовать вложенный класс, как если бы он был компонентом охватывающего класса. Эта техника позволяет избежать создания глобального пространства имен программы (множества имен классов, непосредственно доступных другим программным компонентам), но проблемы остаются.

17.9. Дальнейшее чтение

1. J. Roger Hindley and Jonathan P. Seldin: Introduction to Combinators and λ-Calculus, London Mathematical Society Student Texts, Cambridge University Press, 1986.
   Классическое руководство по лямбда-исчислению и теории комбинаторов (служит непосредственным базисом некоторых языков программирования). Математический текст, не ориентированный на специалистов по информатике. Замечательно ясный текст, определяет все необходимые концепции.
2. Chris Hankin: An Introduction to Lambda Calculi for Computer Scientists, King's College Publications, London, 2004.
   Написана для специалистов по информатике.

17.10. Ключевые концепции данной главы

- Многие программные схемы получают преимущества от механизма упаковки методов в объекты и хранения их для последующих вызовов. Соответствующие конструкции в языке могут быть названы «агентами», другие используемые термины – «делегаты» (в языке C#) и «замыкания».

- Агент, обертывающий метод, может рассматриваться как любой другой объект, например, он может быть присвоен переменной и передан в любую часть программной структуры для вызова метода. Он может быть вызван в любое время через компонент, применимый ко всем агентам, который включает вызов ассоциированного метода, но контексту, где вызывается метод, нет необходимости знать, и обычно он и не знает, что за метод вызывается.
- Агенты могут иметь любое число «открытых» операндов, соответствующих связанным переменным лямбда-выражения. Открытые операнды могут включать некоторые или все аргументы, так же как и цель. Закрытые аргументы (те, что не открыты) специфицируются при задании определения агента. Открытые операнды должны поставляться в форме кортежа при каждом вызове агента.
- Агенты могут быть определены на основе существующего метода. Достаточно указать значения только закрытых операндов, если они есть. Чтобы избежать определения нового метода, когда его нет среди уже существующих, можно определить манифестный агент, написав непосредственно код метода в точке определения агента.
- В языках программирования, не поддерживающих агентов или похожие механизмы, передача функций как данных требует использования многих обертывающих классов, или методов, передаваемых как аргументы, или указателей методов. Эти решения менее удобны, а в последнем случае и менее безопасны.
- Теория лямбда-исчисления предоставляет нужную основу для понимания агентов.
- Лямбда-выражение включает связанные переменные и определяет выражение (которое само может быть лямбда-выражением), включающее возможно связанные переменные, так же как и другие переменные, называемые свободными. Выражение представляет функцию. Применение функции к аргументам сводится к подстановке каждого аргумента для каждого вхождения соответствующей связанной переменной. В результате этого процесса, называемого бета-редукцией, создается новое выражение.
- Связанные переменные лямбда-выражения являются произвольными именами. Они могут быть изменены при условии, что не создаются конфликты имен, в частности, конфликты с именами свободных переменных. Процесс переименования называется альфа-преобразованием (альфа-конверсией).
- Каррингом функции из n аргументов называется специализация (задание значений) m аргументов ($1 \leqslant m \leqslant n$). В результате карринга функция из n аргументов трансформируется в функцию из $n - m$ аргументов.

Новый словарь

Agent	Агент	**Alpha-conversion**	Альфа-преобразование
Beta-reduction	Бета редукция	**Church-Rosser property**	Свойство Черча – Россера
Closed operand	Закрытый операнд	**First-class citizen**	Граждане первого класса
Closure	Замыкание	**Lambda calculus**	Лямбда-исчисление
Inline agent	Манифестный агент	**Many Little Wrappers pattern**	Образец «много маленьких оберток»
Lambda expression	Лямбда-выражение	**One-Song-Artist class**	Класс «певец одной песни»
Nested class	Вложенный класс	**Partial evaluation**	Частичное вычисление
Open operand	Открытый операнд	**Substitution (of a**	Подстановка
Operand	Операнд		
Prototype (C, C++)	Прототип (C, C++)		

	variable in an expression)	(переменной в выражение)

17-У. Упражнения

17-У. 1. Словарь

Дайте определения терминам словаря.

17-У. 2. Карта концепций

Добавьте новые концепции в карту, построенную в предыдущих главах.

17-У. 3. Класс интегрирования без агентов

Смотри соответствующий раздел «Время программирования».

17-У. 4. Объекты итераторы

Спроектируйте механизм итерирования, не использующий агентов, но основанный на классе *LINEAR_ITERATOR*, описывающем объекты, которые допускают итерирование специальной операцией на линейных структурах, подобных списку.

17-У. 5. Итератор, стреляющий себе в ногу

(Это мазохистское упражнение просит вас нарушить все принципы методологии просто для того, чтобы поразмышлять о возникающем беспорядке)

Работая с потомками класса *LINEAR*, такими как *LINEAR_LIST*, используйте процедуру *do_all* с агентом в качестве аргумента, представляющего метод, который, нарушая явное предписание, заданное заголовочным комментарием, изменяет структуру. В результате *do-all* может закончиться неуспехом или даст несогласованный результат. С помощью отладчика, если необходимо, проанализируйте точные обстоятельства, ведущие к отказу.

17-У. 6. Ручная оптимизация

Перепишите *do_all* итератор *LINEAR* так, чтобы он не применял манифестный кортеж как аргумент call, а использовал бы кортежную переменную t, которая заполнялась бы значениями перед каждым вызовом. *Подсказка* 1: вначале создайте объект-кортеж, затем присвойте значение. *Подсказка* 2: перечитайте разделы о свойствах кортежа, особенно тегах.

17-У. 7. Посещение с агентами

Рассмотрите существующее множество классов, например, подмножество классов Traffic. Предположим, что программист может написать операцию *visit*, имеющую вариант для каждого из классов. Эти версии принимают целевой объект как аргумент. Цель упражнения состоит в определении компонента *apply*, который применяет подходящую *visit* операцию к любому такому объекту, переданному как аргумент без знания специфического типа (компонент *apply* может объявить этот аргумент имеющим тип *ANY*).

Не разрешается модифицировать ни один из существующих классов или их потомков. Образец «Посетитель» неприменим, так как вы не можете предполагать, что классы являются потомками класса *VISITOR*.

Покажите, что желаемую цель можно достичь, используя агенты. *Подсказка*: следуйте модели итераторных классов, определенных в этой главе.

Решение, а не только подсказку, можно найти в статье, которая посвящена компонентизации *VISITOR,* цитируемой при обсуждении проблемы.

17-У. 8. Проблема Остановки с агентами

Спроектируйте более выразительное доказательство Проблемы Остановки, которое не использует никаких файлов, каталогов или строк, представляющих тексты программ. Работайте с программными элементами, передаваемыми как агенты.

17-У. 9. Обращение карринга

Отмечалось, что карринг является взаимно обратной функцией. Напишите сигнатуру и определение функции *uncurry*, которой передается функция с одним аргументом *f'*, чей результат – функция с одним аргументом. Функция *uncurry* должна возвращать функцию с двумя аргументами *f*, такую, что *f'= curry(f)*.

17-У. 10. Условие бета-редукции

Покажите, что условие бета-редукции: $[\lambda\, x\colon X \mid exp](e)$ – «не должно быть свободных переменных *e*, появляющихся связанными в *exp*», сильнее, чем это фактически требуется для сохранения неформальной семантики редукции – применения функции к аргументам. Спроектируйте менее строгое, но все еще корректное условие.

17-У. 11. Условие альфа-преобразования

Покажите, что условие альфа-преобразования: $e \triangleq \lambda\, x\colon X \mid exp$ в $\lambda\, y\colon X \mid exp$ [x:= y] – «нет ни свободных, ни связанных вхождений y в *e*», сильнее, чем это фактически требуется для сохранения неформальной семантики редукции – применения функции к аргументам. Спроектируйте менее строгое, но все еще корректное условие.

18

Проектирование, управляемое событиями

Кто в ответе?

В используемом до сих пор стиле программирования порядок выполнения операций определяла программа. Он следовал из ее собственного сценария, определяемого управляющими структурами: последовательностью, условным оператором и оператором цикла. Внешний мир мог сказать свое слово – через интерфейс пользователя, доступ к базе данных, другой ввод, воздействующий на условия, циклы, динамическое связывание. Но последнее слово оставалось за программой, которая решала, где вычислять эти условия.

В этой главе займемся исследованием другой схемы, где программа больше не задает непосредственно последовательность операций, а предоставляет множество *сервисов* (служб), готовых к включению в ответ на *события*, которые могут возникать в результате событий: нажатия пользователем кнопки, обнаружения сенсорным датчиком изменения температуры, сообщения, пришедшего на коммуникационный порт. В любой момент времени следующее событие определяет, какая служба будет востребована. Сразу же после того, как сервис выполнит свою функцию, программа возвращается в состояние ожидания событий.

Такая схема, управляемая событиями, требует подходящей инициализации: перед тем, как начнутся настоящие действия, должна быть фаза установки, регистрирующая службы в соответствии с типами возникших событий.

Такой архитектурный стиль – в конечном счете, еще одна управляющая структура, которую следует добавить к предыдущему каталогу структур, – известна также под именем «**издатели-подписчики**», метафора, отражающая возможное разделение ролей между элементами ПО.

- Некоторые элементы, *издатели*, могут включать события во время выполнения.
- Некоторые элементы, *подписчики*, подписываются на определенные типы событий, указывая, какие службы должны вызываться в ответ на возникновение того или иного события.

Эти роли не являются исключительными, поскольку некоторые подписчики могут включать собственные события. Заметьте, «событие» является программной концепцией: даже когда события возбуждаются вне ПО – нажатие кнопки мыши, измерение датчика, прибытие сообщения, – для того чтобы быть обработанными, они должны транслироваться в программные события. ПО может включать свои собственные события, не связанные с внешними воздействиями.

Программирование, управляемое событиями, применимо во многих различных областях. Оно с успехом применяется в графическом интерфейсе пользователя – GUI, что и станет нашим первым примером.

18.1. Управляемое событиями GUI-программирование

Старый добрый ввод:

Рис. 18.1.

До появления GUI программы вводили данные, приходящие от некоторого последовательного посредника, например, программа могла читать последовательность строк, обрабатывая каждую из них соответствующим образом:

```
from
    read_line
    count := 0
until
    exhausted
loop
    count := count + 1
        -- Сохранить last_line в позиции count в Result:
Result [count] := last_line
    read_line
end
```

Здесь *read_line* пытается читать следующую строку ввода, сохраняя ее в *last_line*, а значение *exhausted* принимает значение *true*, если при выполнении *read_line* нет больше строк для ввода.

При такой схеме **управление остается за программой**: она решает, когда ей нужен очередной ввод данных. Остальной мир — здесь файл или пользователь, печатающий строки на терминале, – должен обеспечить затребованный программой ввод.

Современный интерфейс

Добро пожаловать в современный мир. Если программа использует GUI, то пользователю на каждом шаге предоставляется выбор, что же он хочет делать, включая выбор, не связанный

с программой, поскольку он может переключиться на работу в другом окне с другой программой, например, послать письмо по электронной почте.

Рассмотрим приведенный далее снимок экрана. Он иллюстрирует стек переполнения, возникшего из-за бесконечной рекурсии при выполнении программы в EiffelStudio. Сам пример не имеет значения – все современные среды разработки, использующие GUI, похожи. Интерфейс пользователя включает *элементы управления*: текстовые поля, кнопки, меню, таблицы и другие.

Ожидается, что пользователь выполняет некоторые действия по вводу данных, а наша программа соответствующим образом реагирует на них. Действиями могут быть печать текста в текстовом окне, щелчок по кнопке, выбор пункта меню.

Но какое действие будет первым, и случится ли вообще хоть что-нибудь?

Мы не знаем.

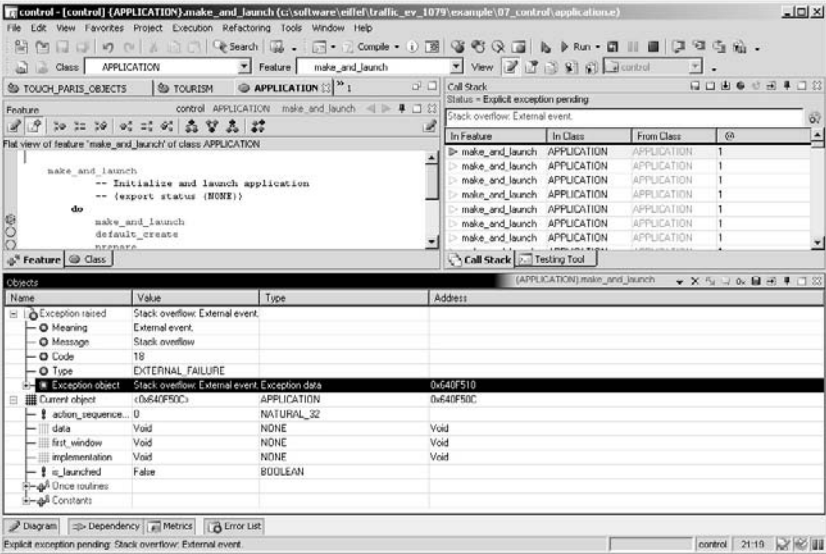


Рис. 18.2. Интерфейс программы

Конечно, мы могли бы использовать большой оператор *if … then … elseif … end* или конструкцию выбора со многими ветвями, перечисляя все возможности:

```
inspect
    user_action
when "Нажата кнопка Stop" then
    "Завершить выполнение"
when "Введен текст в поле Class Name" then
    "В соответствующее окно (слева вверху) вывести текст класса, имя которого
    задано в текстовом поле"
when    … Другие ветви …
end
```

Но это не спасает от всех проблем, с которыми уже приходилось встречаться при обсуждении динамического связывания. Во-первых, программный код в таком случае громоздок, неуклюж и сложен. Во-вторых, что еще хуже, он чувствителен к изменениям. Нам требуется более стабильная и более простая архитектура, не требующая обновлений каждый раз при появлении нового элемента управления.

Проектирование, управляемое событиями («издатели-подписчики»), предлагает в такой ситуации совсем другую схему.

Эту схему можно отобразить как один из экспериментов, проводимых в ядерной физике (смотри следующий рисунок), в котором поток частиц достигает экрана с небольшим отверстием и наблюдается картина, возникающая по другую сторону экрана.

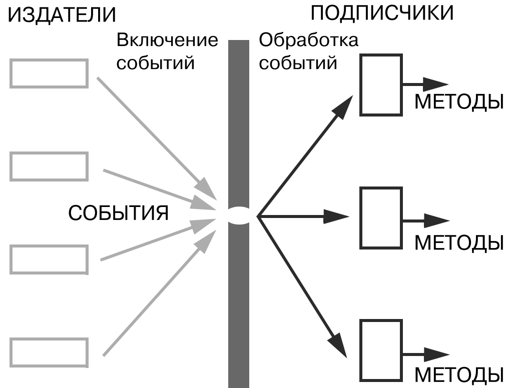


Рис. 18.3. Включение и обработка событий

Стиль «Издатели-подписчики» полезен в различных областях приложения: GUI-программирование – это просто один из примеров. Другие примеры:

- сети передачи данных, где возможна широковещательная передача, – один передатчик и много приемников;
- управление процессом. Сюда входят программные системы, управляющие производственными процессами. В таких системах данные поступают от многочисленных датчиков, следящих за температурой, давлением и другими характеристиками процесса. Возникающие в результате события обрабатываются подготовленными элементами ПО.

18.2. Терминология

Для рассматриваемого стиля программирования важно тщательно определить концепции, обращая, как обычно, внимание на различение типов и их экземпляров.

События, издатели и подписчики

Определение: событие

Событие – это операция периода выполнения, производимая элементом ПО. В результате события некоторая информация становится доступной для потенциального использования другими элементами ПО, не указанными в операции.

Это определение высвечивает отличительные свойства событий.

- Событие сопровождается некоторой информацией: при щелчке мыши должна указываться позиция курсора, при изменении температуры – старая и новая температуры.
- Всегда включается информация о том, какое событие произошло:
  - 5-го августа 1492 года Христфор Колумб отправился в плавание;
  - 5 минут назад (менее известное событие) я щелкнул левую кнопку моей мыши.
- Обычно информации больше – куда отправился Колумб, каковы координаты мыши. Но иногда достаточно знать, что событие произошло, например, что истек срок ожидания.
- Определенные элементы ПО могут использовать эту информацию.
- Во всех случаях важно то, что само событие не знает своих получателей (писатель не знает читателей). В противном случае этот механизм не отличался бы от механизма вызова методов, подобно вызову *x.f (a, b, c)*, который удовлетворяет всем другим свойствам определения – поставляет информацию (a, b, c), доступную элементу ПО (методу f). Но когда вызывается метод, то явно указывается адресат вызова. При вызове событий все не так – информация посылается в междадресное пространство, где любой элемент может использовать ее в интересах своей работы.

Помните, что для наших целей событие – это операция ПО. Могут существовать внешние события – действия пользователя и прочее, – приводящие к возникновению программных событий, но и сама программа может создавать события в процессе работы.

Рассмотрим связанную с этими понятиями терминологию, неформально уже встречавшуюся.

Определения: включение, публикация, издатель, подписчик

> Включением (или публикацией) события является его выполнение. Элемент ПО, включающий событие, – издатель. Элемент ПО, использующий информацию события, – подписчик.

Один и тот же элемент ПО может быть издателем и подписчиком, в частности, благодаря общей для подписчиков схеме – включать собственное событие как реакцию на возникновение пришедшего к нему события.

В литературе встречаются синонимы приведенных выше терминов: подписчики называются «*наблюдателями*», отсюда образец «Наблюдатель», изучаемый в этой главе. Говорят также о «*слушателях*» события. Для «издателей» также используется синоним, вытекающий из подобной риторики – «*предмет наблюдения*».

Аргументы и типы событий

Нам необходимо имя для информации, приходящей – в соответствии с определением – с любым событием.

Определение: аргумент

> Информация, связанная с событием, состоит из **аргументов** события.

Термин «аргумент» указывает на сходство с аргументами метода. Продолжая это сходство, будем предполагать, что аргументы сгруппированы в упорядоченный список, подобно аргументам в вызове *x.f (a, b, c)*. Как и для методов, список может быть пустым, например, в случае события, указывающего на истечение срока ожидания.

Как подписчики обнаруживают, что произошло интересующее их событие? Одна модель – опрос (повторяющаяся проверка). Это напоминает ситуацию с подписчиком журнала, который в день издания ходит к почтовому ящику, чтобы проверить, не пришел ли журнал. Вторая модель – уведомление: при включении события об этом уведомляются все подписчики.

Модели для информации, распределенной в сети Интернет, классифицируются как «притяжение» (ожидание пользователей, заинтересованных в информации) и «проталкивание» (рассылка информации пользователям).

Модель уведомления – более гибкая, и далее будем предполагать именно ее. Она может работать при условии, что подписчики заранее явно проявили свой интерес. Дело обстоит так же как с подписчиками журнала: на журнал надо заранее подписаться. Но на что можно подписаться? На событие подписаться нельзя, поскольку, по определению, это операция, выполняемая в ходе работы программы: до его появления оно не существует, а после – уже поздно.

То, что нужно знать подписчикам, это тип события, описывающий возможные события, разделяющие общие характеристики. Например, все щелчки левой кнопки мыши – это события одного типа, но отличающиеся от конкретного события нажатия клавиши. Понятие типа события играет центральную роль в проектировании, управляемом событиями, и будет представлять центральную абстракцию в нашем поиске хорошей ОО-архитектуры.

Все события одного типа имеют один и тот же список аргументов. Например, список аргументов для события – щелчок левой кнопки мыши – включает координаты мыши в момент щелчка – два целых числа. Во многом концепция заимствована у методов, обладающих сигнатурой – списком типов аргументов. Процедура *print* (*v*: *VALUE*; *f*: *FORMAT*) имеет сигнатуру [*VALUE*, *FORMAT*] – список типов. Это понятие расширяется на типы события.

Определения: тип события, сигнатура

Любое событие принадлежит определенному типу события.

Все события данного типа события имеют одну и ту же сигнатуру.

Например:

- сигнатура события «изменение температуры» может быть [*REAL*, *REAL*], чтобы представлять старое и новое значения;
- событие «левый щелчок мыши» может иметь сигнатуру [*INTEGER*, *INTEGER*].

Можно также взять событие «одиночный щелчок мыши» с третьим компонентом сигнатуры, указывающим, какая кнопка была нажата. В библиотеке EiffelVision применяется вариант, в котором добавлен еще один аргумент, показывающий, остается ли нажатой кнопка, – он полезен (особенно в играх) для джойстика и экзотичных устройств указателей;

- хотя можно было бы определить тип события для каждой клавиши на клавиатуре, более удобно использовать один тип события «клавиша нажата» с сигнатурой [*CHARACTER*], где аргумент задает код клавиши;
- для событий без аргументов, например, «исчерпано время ожидания», сигнатура пуста, как у методов без аргументов.

Всякий раз, когда издатель включает событие, он должен обеспечить значение каждого аргумента (если они есть): координаты мыши, код клавиши, температуры. И здесь наблюдается полная аналогия с вызовом метода, где при каждом вызове задаются фактические аргументы.

Термин «тип события» может предполагать еще одну аналогию, где каждый тип соответствует типам ОО-программирования (классам с возможно родовыми параметрами), а каждое событие соответствует экземпляру класса (объектам). Но сравнение типов событий с программами – более показательное, тогда появление события данного типа соответствует вызову программы.

При таком подходе событие – это не объект, а тип события – это не класс. Вместо этого можно ввести общее понятие для всех типов событий: класс, называемый ниже *EVENT_TYPE*, а конкретный тип события, например, «щелчок левой кнопки мыши» (абстракция левых щелчков – вроде того, что я в прошлый понедельник, будучи в расстроенных чувствах, щелкнул в ответ на запрос «Delete all?») рассматривать как объект. Как всегда, когда вы раздумываете, не ввести ли класс, критерием является «Можно ли эту абстракцию данных наполнить смыслом, определив множество хорошо понимаемых операций, применимых ко всем объектам класса?». В данном случае:

- если бы мы решили создавать класс для типа события, его экземплярами были бы события одного типа, но у них не было бы полезных компонентов. Более точно, события имели бы собственные данные – аргументы, но они нуждались бы только в запросах для получения доступа к этим аргументам; команд здесь не было бы;
- в противоположность такому подходу: если рассматривать конкретный тип события как объект, то появляется несколько хорошо определенных команд и запросов – включить событие с заданным набором аргументов, подписать данного подписчика на этот тип события, удалить подписчика из списка подписчиков, перечислить всех подписчиков, подсчитать число включений события данного типа и так далее. Это богатое множество компонентов характеризует полезный законный класс.

Не рассматривать каждое событие как объект полезно и с позиций производительности. Обычно во время выполнения создается большое число событий – каждое малое перемещение курсора включает событие, так что следует избегать создания всех соответствующих объектов. Это не освобождает нас от нагрузки, поскольку аргументы каждого события, представленные кортежем, должны быть записаны. В хорошей библиотеке GUI производительность улучшается за счет того, что для последовательности близких событий можно использовать один кортеж вместо десятков сотен.

Полезно иметь термины для действий подписчиков с типами событий и событиями.

Определения: подписать, зарегистрировать, обработать, захватить

Элемент ПО может стать подписчиком некоторого типа событий, **подписавшись** (**зарегистрировавшись**) на него. После регистрации элемент будет получать уведомления о всех возникающих событиях этого типа, так что он может получить аргументы и выполнить в ответ специфические действия.

Когда элемент получает уведомление о событии, на которое он подписан, он **обрабатывает** (или **захватывает**) событие, выполняя зарегистрированное действие.

Хотя регистрацию (дерегистрацию) можно выполнить в любой момент, общепринято иметь фазу инициализации, расставляя подписчиков по местам, после чего уже выполняет-

ся главный этап выполнения, на котором издатели включают события, а подписчики их обрабатывают.

При регистрации подписчик задает некоторое действие, выполняемое в ответ на возникновение события данного типа. У действия должен быть способ получения аргументов события. Очевидный путь достижения цели – это указание при регистрации метода, чья сигнатура соответствует сигнатуре типа события. Тогда возникновение события данного типа будет причиной вызова метода, обрабатывающего событие, где аргументы события играют роль фактических аргументов вызова.

Теперь у нас есть полная картина того, как работает программа, управляемая событиями.

Схема управления событиями

E1 Некоторые элементы, *издатели*, позволяют остальной системе узнать, какие *типы событий* они могут включать.

E2 Некоторые элементы, *подписчики*, заинтересованы в *обработке* событий определенных типов. Они *регистрируют* соответствующие действия.

E3 В любой момент издатель может *включить* событие. Это приведет к выполнению действий, зарегистрированных подписчиками для события данного типа. Эти действия могут использовать аргументы события.

В примере GUI:

E1 Издателем является некоторый элемент ПО, который следит за устройствами ввода и включает события при определенных обстоятельствах, например, при нажатии клавиш клавиатуры или кнопок мыши. Обычно нет необходимости писать такое ПО, поскольку оно является частью библиотеки GUI - EiffelVision для Eiffel, Windows Forms для .NET, Swing для Java.

E2 Подписчиком является любой элемент, которому требуется обработать GUI-события. Он регистрирует методы, выполняемые в ответ на события. Например, метод, сохраняющий файл, можно зарегистрировать для события щелчок мыши на кнопке с надписью «OK» в диалоге по сохранению файла.

E3 Если во время выполнения пользователь щелкнет по кнопке OK, то это станет причиной выполнения метода – или методов, – зарегистрированных для данного типа события.

Важное свойство этой схемы, проиллюстрированное на последнем рисунке: выделение двух сторон, участвующих в создании и обработке события, – подписчики и издатели ничего не знают друг о друге. Более точно, определение «события» требует, чтобы подписчики не знали других подписчиков. Остальное – дело методологии, и мы увидим, как различные архитектурные решения достигают результата по отношению к этому критерию.

Ясное понимание различий

Возможно, вы полагаете, что различие между типом события и самим событием очевидно, но в литературе эти понятия зачастую путаются, из-за чего простые вещи кажутся сложными. Это предупреждение должно помочь вам при изучении различных механизмов программирования, управляемого событиями.

Следующий текст представляет часть документации .NET от Microsoft, представляющей обработку событий, чьи концепции отражены в языках C# и Visual Basic .NET.

Обзор событий

События имеют следующие свойства.

1. Издатель определяет, когда событие включается; подписчики определяют, какие действия будут выполняться в ответ на событие.
2. Событие может иметь много подписчиков. Подписчик может обрабатывать события, приходящие от многих издателей.
3. События, не имеющие подписчиков, никогда не вызываются.
4. События широко используются для сигнализации о действиях пользователя, таких как щелчки по кнопкам или выбор в меню в графическом интерфейсе пользователя.
5. При наличии множества подписчиков вызов обработчиков события синхронизируется. Для асинхронного вызова смотри следующий раздел.
6. События могут использоваться для синхронизации потоков.
7. События в библиотеке классов .NET Framework основаны на делегате EventHandler и базовом классе EventArgs.

Здесь один и тот же термин «событие» в разных контекстах имеет разный смысл: иногда речь идет действительно о событиях, иногда о типах событий, иногда об обоих понятиях одновременно. Думаю, что вы согласитесь с моей интерпретацией. В частности:

- невозможно подписываться на события (пункты 1, 5). Как мы видели, событие не существует, пока оно не будет возбуждено. Подписчик подписывается на *тип события*, уведомляя, что он хочет получать уведомление о каждом событии этого типа, когда оно произойдет во время выполнения;
- в пункте 7 говорится о свойствах *классов*, описывающих *типы* событий. Заметьте, что в .NET каждый тип события объявляется как класс. Такой класс должен наследовать от класса EventHandler, представляющего классы, которые называются делегатами (delegate) – последние обеспечивают механизм, подобный агентам. Еще один класс – EventArgs – является родительским классом для классов, задающих аргументы события;
- пункт 3 звучит загадочно, пока не осознаешь, что имеется в виду следующее: «Если тип события не имеет подписчиков, включение события этого типа не дает никакого эффекта». Все это описывает внутреннюю оптимизацию: обнаружив, что тип события не имеет подписчиков, механизм события удаляет избыточно возбужденные события, что в .NET влечет к созданию объекта для каждого события.

Возможность непонимания особенно ярко проявляется в двух местах.

- В пункте 2 говорится: «Подписчик может обрабатывать множественные события от множественных издателей». Из этого комментария можно сделать вывод, что речь идет о сложной параллельной схеме вычислений, где подписчик захватывает события из различных мест одновременно, но реально это просто означает, что подписчик может регистрировать несколько типов событий и каждый издатель может включать события разных типов.
- В пункте 5 устанавливается, что для события, имеющего множество подписчиков, вызов обработчиков события синхронизируется. Предложение просто пытается сказать, что когда множество подписчиков зарегистрировали один и тот же тип события, они обрабатывают соответствующие события синхронно.

Так что при изучении схем управления событий проверяйте, о чем идет речь – о событиях или о типах событий, и убедитесь (это одно из наших очередных наставлений), что в вашей документации по событиям используется правильная терминология.

Контексты

Подписчик при регистрации говорит: «Для события этого типа выполняй это действие». На практике может быть полезно, особенно для приложений GUI, уточнить высказывание: «Для события этого типа, встречающегося в данном контексте, выполняй это действие». Например:

- «если пользователь нажал левую кнопку мыши на кнопке OK, сохрани файл»;
- «если мышь находится в этом окне, измени цвет границ на красный»;
- «если датчик показывает температуру выше 25° С, включи сигнал тревоги».

В первом случае «контекстом» является элемент управления, а событием – «щелчок мыши». Во втором контекст – окно, событие – появление мыши; в третьем контекст – датчик, событие – превышение режима.

Для GUI-программирования контекстом обычно является элемент управления. Как показывает последний пример, понятие контекста более общее; контекст может быть любым условием с булевским значением. Примеры GUI являются специальным случаем, где булевское условие задает свойство, такое как «курсор установлен на этой кнопке» или «курсор находится в этом окне». Вот общее определение:

Определение: контекст

При управлении событиями **контекст** представляет булевское выражение, задаваемое подписчиком в момент *регистрации*, но вычисляемое в момент *включения*, так что регистрируемое действие будет выполняться только, когда контекст принимает значение **True.**

Понятие контекста нам знакомо по обычному стилю программирования, не связанному с событиями: вспомните итератор, такой как *do_if*, который выполняет действия над всеми элементами структуры, удовлетворяющими некоторому условию.

Это аналогично тому, как контекст позволяет подписчику установить, что его интересуют события данного типа, но необходимо, чтобы в момент включения выполнялось определенное условие.

Без понятия «контекст» можно обойтись, если включать ассоциированное условие в само регистрируемое действие, например:

```
if "Курсор на значке Exit" then
    "Выполнить предусмотренные действия"
end
```

Удобнее отделять условие, задавая его вместе с типом события и действием.

18.3. Требования «Публиковать-подписаться»

Установив концепции, будем теперь заниматься поиском общего решения проблемы, проектируя архитектуру управления событиями. Начнем с ограничений, которым должно удовлетворять хорошее решение.

Издатели и подписчики

При проектировании архитектуры, поддерживающей парадигму «Публиковать-подписаться», следует рассмотреть следующие ограничения.

- Издатели не должны знать, кто является подписчиком. Они включают события, но в соответствии с определением события не знают, кто может их обрабатывать. Типичным случаем издателя является библиотека GUI. Методы библиотеки знают, как обнаружить событие, инициированное пользователем, такое как «щелчок кнопки», но они ничего не должны знать о конкретных приложениях, реагирующих на эти события, и о том, как они реагируют. В приложении нажатие кнопки может сигнализировать о начале компиляции, запуске процесса оплаты, выключении завода или запуске ракеты. Но все это для библиотеки GUI – просто щелчок кнопки.
- Любое событие, включаемое одним издателем, может поставляться нескольким подписчикам. Изменение температуры в системе управления завода может отражаться в местах, «наблюдающих» за этим типом события. Данные могут отражаться на обычном и на графическом дисплее, в службе безопасности, включающей определенные действия при нарушениях режима, в записях изменений в базе данных.
- Подписчикам нет необходимости знать издателей. Это более строгое, но часто желательное требование. Подписчики знают о типах событий, на которые они подписываются, но не должны знать, откуда приходят события. Помните, что одна из целей проектирования, управляемого событиями, состоит в обеспечении гибкой архитектуры, позволяющей включать разных издателей и разных подписчиков, возможно, написанных разными людьми и в разные времена.
- Желательно иметь возможность во время выполнения как проводить регистрацию, так и отменять ее. В обычной схеме регистрация выполняется в фазе инициализации приложения, где устанавливаются его параметры до начала «настоящего» выполнения. Но это не является обязательным требованием – гибкость может быть полезной.
- Должно быть возможным задавать события, зависящие или не зависящие от контекста. Мы видели полезность связывания события с контекстом, но решение должно обеспечивать возможность не придумывать искусственный контекст и просто подписаться на событие независимо от того, где оно случилось.
- Связывание издателей и подписчиков должно выполняться с минимальными усилиями. Схему с событиями часто требуется добавить в уже существующее приложение. Для связывания сторон требуется добавить некоторый код, часто называемый «склеивающим кодом»: чем меньше клея – тем лучше.

Последнее требование является критическим для качества системной архитектуры, особенно когда целью является построение пользовательских интерфейсов: не должно быть так, чтобы проектирование ядра зависело от особенностей интерфейса. Это наблюдение непосредственно приводит к нашим следующим понятиям – модели и облику.

Модель и облик

При проектировании интерфейса мы не только не должны различать подписчиков и издателей, но и различать два дополняющих аспекта приложения.

Определения: модель и облик программной системы

Модель (называемая также *бизнес-моделью*) является той частью программной системы, которая обрабатывает данные, представляющие информацию прикладной области.

Облик – это представление части этой информации при взаимодействии системы с внешним окружением: человеком – пользователем системы, материальными устройствами, другим ПО.

В этом определении термин «прикладная область» используется в общепринятом смысле, как техническая область, в интересах которой создается и работает приложение. Для платежной системы предприятия прикладной областью является штат компании, для ПО, управляющего полетом, таковой является система управления воздушным сообщением.

Модель является частью ПО – частью, имеющей дело с прикладной областью. Для платежной системы это та часть, которая обрабатывает информацию о служащих, их часах работы, начисляет зарплату, обновляет базу данных. Для системы управления полетом – прокладывает маршрут, вычисляет времена, авторизацию и прочее. Можно сказать, что модель – это часть, выполняющая «настоящую» работу, независимо от взаимодействия с пользователями ПО и остальным миром.

Понятие «бизнес-модель» является более точным, но мы обычно предпочитаем говорить просто «модель». Одна из причин в том, что термин «бизнес» порождает неверные ассоциации (управление компанией, финансами), исключая такие области, как обработка текстов или управление полетами.

Облик задает представление информации, обычно на входе и выходе. Обликом является GUI: например, система управления полетом имеет интерфейс, позволяющий контролировать следование запланированной траектории, вводить нужные команды.

Обычно программа предназначена для одной – возможно, весьма широкой – прикладной области, но обликов у программы может быть несколько. Хорошей практикой является рассмотрение программы с разных точек зрения. При наивном проектировании небольших программ не уделяется должного внимания этой проблеме. Но для серьезных систем необходимо планировать *несколько обликов*, таких как:

- GUI-облик (обычно несколько);
- Web-облик («WUI»), позволяющий использовать систему через Web-браузер;
- текстовый интерфейс – для ситуаций, когда графическая поддержка не нужна или невозможна;
- интерфейс, основанный на batch-файлах, где подготовлен сценарий, заготовлены исходные данные и вывод пишется в заданное место. Это особенно полезно для интерактивных систем. Интерактивное тестирование трудно, оно требует присутствия людей, проводящих долгие сессии, пытаясь проверить различные комбинации. Вместо этого можно подготовить коллекцию сценариев (обычно записываемых во время сессии с участием человека), а затем выполнять их без участия человека;
- облики, обеспечиваемые другими программами, которые выполняются локально; их функциональность доступна через API;
- облики Web-сервисов, обеспечиваемые программами, которые выполняются на других компьютерах; их функциональность доступна через Web-направленный API (эти сервисы требуют специальной техники, такой как протокол SOAP).

Вначале обычно достаточно одного облика. Вот почему типичной ошибкой проектирования является построение системы, где модель и облик сложно связаны. Затем, когда понадобятся другие облики, приходится прикладывать массу усилий по перепроектированию системы. Во избежание этого общим принципом должно быть разделение модели и ее обликов уже на начальных этапах проектирования системы.

Почувствуй методологию

Принцип разделения модель/облик

При проектировании архитектуры программной системы сводите к минимуму взаимодействие элементов модели и элементов облика.

Если мы используем архитектуру, управляемую событиями, то это правило хорошо сочетается с четким разделением издателей и подписчиков. Как издатели, так и подписчики взаимодействуют с обликом, но не связанными между собой способами.

- Издатели включают события, которые могут непосредственно изменить облик, обычно в малой мере. Например, курсор может изменить форму при перемещении в другое окно, нажатая кнопка может выглядеть иначе, чем кнопка в обычном состоянии.
- Подписчики захватывают события (тех типов, на которые они подписались) и обрабатывают их. Обработка может обновить облик.

Заметьте, два разделения – издатели-подписчики и модели-облики – взаимно ортогональны. Как издатели, так и подписчики могут взаимодействовать как с моделью, так и с обликами, как это можно видеть на примере системы обработки текстов.

- для издателей включать событие может возникать из-за изменения облика – пользователь передвинул мышь или нажал на кнопку – или из-за модели, когда, например, система проверки орфографии обнаружила неверно написанное слово и подчеркнула его.
- Обработка события подписчиком часто становится причиной модификаций как в модели, так и в облике. Например, если пользователь выделил некоторый текст и затем нажал клавишу Delete, то эффект двойной – удаляется часть хранимого текста (модель) и обновляется та часть экрана, где текст отображается (облик).

Модель – облик – контроллер

Для проектирования GUI особый интерес представляет схема «Модель – облик – контроллер» (МОК). Роль третьего элемента – контроллера – состоит в управлении интерактивной сессией. Она может включать создание и координацию обликов.

Каждая из трех частей взаимодействует с двумя другими:

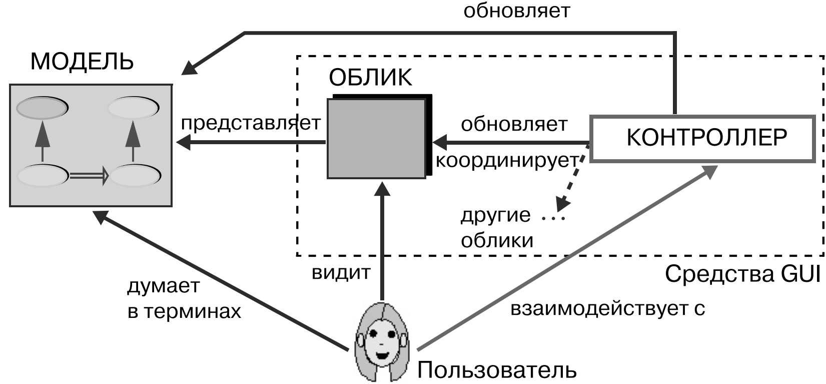


Рис. 18.4. Структура МОК

Присутствие контроллера обеспечивает дальнейшее разделение между моделью и обликами (помните, что обликов может быть несколько). Контроллер управляет действиями пользователя, которые могут приводить к обновлению модели, облика, или того и другого.

Как и ранее, облик обеспечивает визуальное представление модели или части ее.

Проектировщик системы может предполагать, что пользователи понимают модель. Используя текстовый процессор, пользователь обычно знаком со шрифтами, абзацами, разде-

лами. Пользователь, играющий в видеоигру, должен чувствовать космическое пространство и летящие ракеты. Хорошая система позволяет пользователю думать в терминах модели. Хотя то, что я вижу на экране, не более чем несколько пикселей, образующих круг, я думаю об этом как о летящем космическом корабле. Контроллер позволяет мне действовать над такими обликами, например, вращая колесико мыши, увеличивать скорость космического корабля, при этом будет обновляться как модель (изменяются ее атрибуты – скорость, позиция), так и облик, отражающий изменения в визуальном представлении.

Парадигма МОК оказала существенное влияние на скорость распространения графических интерактивных приложений за последние десятилетия. В конце главы мы увидим, что принимая понятие проектирования, управляемого событиями с вытекающими последствиями, можно получить преимущества МОК, но с более простой архитектурой, избегая некоторых отношений, показанных на предыдущем рисунке.

> Пользуясь случаем, дадим несколько полезных советов, связанных с рисунками. В презентациях ПО часто приводятся выразительные диаграммы с многочисленными блоками, связанными стрелками. Одна беда – недостаточно спецификаций, поясняющих семантику. На нашем рисунке используются для этой цели метки, такие как «представляет», «обновляет» и другие. Неименованные стрелки имеют стандартную семантику, задавая отношения наследования или «клиент-поставщик». Рисунок не хуже многих слов, если только это не просто цветовые эффекты. Не поддавайтесь соблазнам бессмысленной графики – явно задавайте точную семантику используемых символов.

18.4. Образец «Наблюдатель»

Прежде чем приступить к определению схемы проектирования, управляемого событиями (по меньшей мере, для примеров, обсуждаемых в этой главе), давайте исследуем образец проектирования «Наблюдатель», который также связан с рассматриваемой нами проблемой.

Об образцах проектирования

Образец проектирования – это стандартизованная архитектура, ориентированная на определенный класс задач. Такая архитектура типично определяется как совокупность классов, которые должны быть частью решения. Для классов указывается их роль, их отношения, кто от кого наследует, кто является чьим клиентом. Образец сопровождается указаниями по настройке классов на решение конкретной задачи. Образцы проектирования появились в середине девяностых годов как способ записи и создания каталога решений, применяемых при проектировании. Они известны также как лучшая практика, которую программисты вырабатывали годами, часто приходя независимо к одинаковым решениям.

Десятки таких образцов – «Наблюдатель» один из них – документированы и широко изучены.

Основы образца «Наблюдатель»

В качестве общего решения управления событиями «Наблюдатель» не вполне хорош – его ограничения будут проанализированы. Но его следует знать по ряду причин.

- Это некоторая классика.
- Здесь элегантно используются преимущества ОО-механизмов, таких как полиморфизм и динамическое связывание.

- Это лучшее, что можно сделать в языках, где нет агентов, универсальности и кортежей.
- Он дает хорошую основу для перехода к более разумному решению, изучаемому далее.

На следующем рисунке приведена типичная архитектура «Наблюдателя». Два общецелевых класса *PUBLISHER* и *SUBSCRIBER* не несут специфики конкретного приложения; *PUBi* и *SUBj* используются для представления классов типичного издателя и подписчика в вашем приложении:

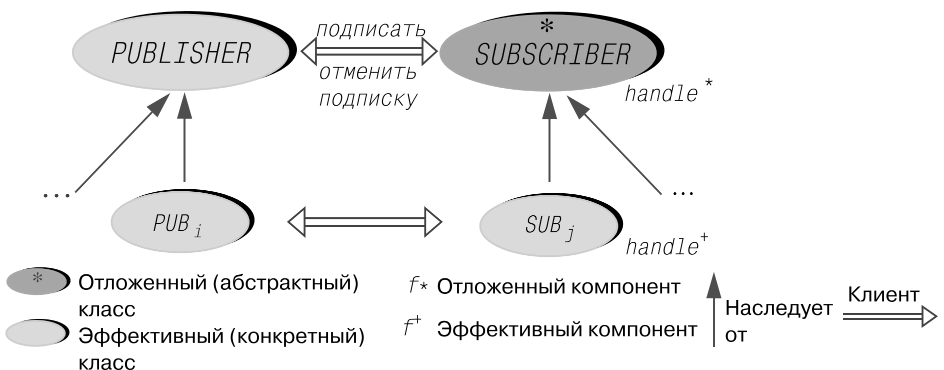


Рис. 18.5. Архитектура образца Наблюдатель

Хотя оба класса, *PUBLISHER* и *SUBSCRIBER,* предназначены для роли предков классов, выполняющих фактическую работу по публикации и обработке событий, только класс *SUBSCRIBER* должен быть отложенным. Отложенная процедура *handle* этого класса будет конкретизирована потомком *SUBj*, где и будет показано, как конкретный подписчик обрабатывает событие. У класса *PUBLISHER* нет необходимости в отложенных компонентах.

Как отмечалось, подписчики *наблюдают* (отсюда имя образца) за издателями, ожидая от них сообщений. Издатели являются субъектом наблюдения. В литературе по образцам используются различные синонимы применяемых нами терминов.

На стороне издателя

Класс *PUBLISHER* определяет свойства типичного издателя, отвечающего за тип события. Процедура *publish* этого класса позволяет включать события этого типа. Главной структурой данных является *список подписчиков*.

```
note
    what: Наблюдаемые подписчиками объекты, публикующие события одного типа
class
    PUBLISHER
feature {SUBSCRIBER} — Status report
    subscribed (s : SUBSCRIBER): BOOLEAN
            — Является ли s подписчиком данного издателя?
        do
            Result := subscribers.has (s)
        ensure
```

```
            present: has (s)
        end
feature {SUBSCRIBER} – Element change
    subscribe (s : SUBSCRIBER)
            – Сделать s подписчиком данного издателя.
        do
            subscribers.extend (s)
        ensure
            present: subscribed (s)
        end
    unsubscribe (s : SUBSCRIBER)
                – Удалить s из списка подписчиков этого издателя.
        do
                subscribers.remove_all_occurrences (s)
        ensure
                absent: not subscribed (s)
        end
    publish (args : LIST [ANY ])                              –Схема аргументов 1
                – Опубликовать событие для подписчиков.
        do
                … Смотри ниже …
        end
feature {NONE} –Реализация
    subscribers: LINKED_LIST [SUBSCRIBER]
            – Подписчики, подписанные на событие этого издателя.
end
```

Процедура *publish* уведомляет всех подписчиков этого события (тип события, за который ответственен издатель), что событие произошло. Мы увидим, как можно написать это проще, после проектирования класса, представляющего типичного подписчика.

Реализация позволяет дважды вызывать *subscribe* для того же самого подписчика, тогда (смотри *publish* ниже) подписчик будет дважды выполнять предписанные действия для каждого события.

В большинстве случаев такой эффект не желателен. Во избежание этого можно было бы обернуть тело *subscribe* в **if not** *subscribed* (*s*) **then** ... **end**, но тогда терялась бы эффективность связного списка, так как требовался бы его обход. Для нашего обсуждения это не критично, но должно учитываться при любом практическом использовании образца; эта проблема является предметом упражнения в конце этой главы.

Кроме атрибута *subscribers*, спроектированного для внутренних целей и, следовательно, закрытого (экспортируемого NONE), остальные компоненты предназначены только для подписчиков, но не для объектов других типов, — по этой причине они экспортируются только классу *SUBSCRIBER* (как вы помните, это означает, что они экспортируются и потомкам класса, которым необходима возможность подписаться или отменить подписку).

Общее правило говорит, что экспорт должен быть выборочным. Класс должен предоставлять компоненты, спроектированные в интересах определенных классов, только этим классам и их потомкам. Лучше быть более строгим, чтобы избежать ошибки использования специализированного компонента объектами класса, не приспособленного к применению компонента. Ослабить ограничение проще, чем усилить.

На стороне подписчика

```
note
    what: "Регистрируемые объекты, обрабатывающие события данного типа"
deferred class
    SUBSCRIBER
feature - Element change
    subscribe (p: PUBLISHER)
                - Подписаться у издателя p.
        do
                p.subscribe (Current)
        ensure
            present: p.subscribed (Current)
        end
    unsubscribe (p: PUBLISHER)
            - Убедиться, что этот подписчик не подписан у p.
        do
            p.unsubscribe (Current)
        ensure
            absent: not p.subscribed (Current)
        end
feature {NONE} - Basic operations
        handle (args: LIST [ANY])                       -Схема аргументов 1
            - Реакция на публикацию события подписанного типа
        deferredend
        end
end
```

О схеме аргументов будет сказано ниже.

Этот класс отложен: любой класс приложения может, если его экземплярам необходимо действовать как подписчикам, наследовать от *SUBSCRIBER*. Будем называть таких потомков «классами подписчиков», а их экземпляры – «подписчиками».

Чтобы подписаться на тип события у соответствующего издателя *p*, подписчик выполняет *subscribe* (*p*). Заметьте, как эта процедура (и аналогично *unsubscribe*) использует соответствующий компонент от *PUBLISHER,* передавая ему для подписки текущий объект. Это еще одна из причин выборочного экспорта в классе *PUBLISHER* – было бы бесполезно для класса подписчика применять *subscribe* от *PUBLISHER* непосредственно. Подписка имеет смысл, только если обеспечивается соответствующий механизм обработки *handle*, приходящий от класса *SUBSCRIBER* (этим объясняется и использование общих имен компонентов в двух классах, что сохраняет терминологию простой и не приводит к недоразумениям, так как компоненты экспортируются только подписчикам).

> Процедура *unsubscribe* удаляет подписчика у соответствующего издателя. Во избежание потерь памяти не забывайте вызывать ее, когда подписчику подписка становится ненужной.

Каждый класс подписчика обеспечивает свою собственную версию обработчика события – *handle*. Основная идея *handle* проста: вызывается требуемая операция, которой передаются аргументы события, если они есть.

Один неприятный момент: необходимо убедиться, что операция получает аргументы правильных типов. Причина в том, что мы пытаемся сделать *PUBLISHER* и *SUBSCRIBER* общими, а потому должны объявить аргументы *args*, представляющие аргументы события как в *publish*, так и в *handle* полностью общего типа: *LIST* [*ANY*]. Но тогда *handle* должна выполнять кастинг, преобразуя *args* к правильному типу и числу аргументов.

Предположим, например, что тип события объявляет два аргумента соответствующих типов *T* и *U*. Мы хотим обработать каждое событие, вызывая метод *op* (*x*: *T; y: U)*. Тогда следует написать *handle* следующим образом:

```
handle (args: LIST [ANY ])                          – Схема Аргументов 1
    – Выполнение операции op над аргументами в ответ на публикацию события.
    do
        if args.count >= 2 and then
            (attached {T } args.item (1) as x) and
            (attached {U } args.item (2) as y)
        then
            op (x, y)
        else
            – Не делать ничего или выдать отчет об ошибке
        end
    end
```

Тест объектов позволит убедиться, что первый и второй элементы списка *args* имеют ожидаемые типы и свяжет их с *x* и *y* внутри **then** ветви

Единственный способ избежать такого тестирования, выполняемого в период выполнения, состоит в специализации *PUBLISHER* и *SUBSCRIBER*, объявив *publish* и *subscribe* с точными типами аргументов, например:

```
publish (x: T ; y : U)              – Схема аргументов 2
```

Аналогично для *handle* в *SUBSCRIBER*. В этом случае теряется общность схемы, поскольку нельзя использовать классы *PUBLISHER* и *SUBSCRIBER* для типов событий с различающимися сигнатурами. Хотя отчасти это дело вкуса, но я рекомендовал бы схему аргументов 2 в случае применения образца «Наблюдатель», поскольку в этом случае ошибки – издатель передал неверные аргументы – будут обнаруживаться на этапе компиляции.

Для метода *handle*, соответствующего схеме аргументов 1, ошибки будут обнаруживаться только на этапе выполнения, когда поздно предпринимать разумные действия. Комментарий говорит, что в этом случае нужно либо ничего не делать (но странно игнорировать ожидаемое событие), либо выдать отчет об ошибке (но ошибка – разработчиков, а сообщение будет выдано конечным пользователям).

При обсуждении теста объектов отмечалось, что этот механизм должен обычно резервироваться для объектов, приходящих из внешнего мира, а не от тех, что находятся под контролем программы, для которых проектировщик ответственен за статическую проверку правильности типов. Здесь публикация и обработка событий принадлежат одной и той же программе, поэтому использование теста объектов не представляется убедительным.

Можно построить решение, безопасное по типам, сделав классы *PUBLISHER* и *SUBSCRIBER* универсальными. Родовым параметром является кортежный тип, представляющий сигнатуру типа события (другими словами – последовательность типов аргументов). Подобное решение появится ниже в заключительной архитектуре «публиковать-подписываться» («Event Library»). Мы не станем разрабатывать его для образца «Наблюдатель», поскольку оно основано на механизмах – типы кортежей, ограниченная универсальность, – недоступных в других языках. Если же вы программируете на Eiffel, то следует использовать заключительную архитектуру, которая лучше образца «Наблюдатель» и доступна в библиотеке классов Eiffel. Однако это хорошее упражнение – улучшить «Наблюдатель», используя эти идеи. Попытайтесь это сделать прямо сейчас, не дожидаясь появления решения на последующих страницах.

Публикация события

Единственная пропущенная часть реализации образца «Наблюдатель» — это тело процедуры *publish* в *PUBLISHER*, хотя, я надеюсь, мысленно вы ее уже написали. Вот где образец выглядит особенно элегантным:

```
publish (args: … Схема аргументов 1 или 2, смотри обсуждение выше …)
            — Публиковать событие для подписчиков.
      do
            — Просить каждого из подписчиков в свою очередь
            — обработать послание:
            from subscribers.start until subscribers.after loop
               subscribers.item.handle (args)
               subscribers.forth
            end
      end
```

Для схемы аргументов 1 *args* принадлежат типу *LIST* [*ANY*]. Для схемы аргументов 2 объявление специфицирует точно ожидаемые типы.

Операторы программы показывают преимущества полиморфизма и динамического связывания: *subscribers* — это полиморфный контейнер, каждый элемент списка может быть разного *SUBSCRIBER*-типа, характеризуемый своим вариантом обработки события *handle*. Динамическое связывание гарантирует, что в каждом случае будет вызываться правильный вариант. Это просто праздник лучших приемов ОО-архитектуры!

Оценка образца «Наблюдатель»

Образец «Наблюдатель» широко известен и используется. Это интересное применение ОО-приемов. Как общее решение проблемы «публиковать-подписаться», он страдает рядом ограничений.

- Как обсуждалось, возникает неприятная дилемма выбора между двумя схемами аргументов, одинаково непривлекательными: с одной стороны, опасная проверка типов аргументов в момент выполнения, чреватая ошибками, с другой — специфические, квазиидентичные классы *PUBLISHER* и *SUBSCRIBER*, задающие сигнатуру для каждого типа события.
- Подписчики непосредственно подписываются у издателей. Это и есть причина нежелательной связи между двумя сторонами. Подписчики не должны знать, какая часть приложения или какая библиотека включает события. Фактически, пропущен еще

один участник процесса – брокер – посредник между двумя сторонами. Более фундаментальная причина состоит в том, что пропущена ключевая абстракция – тип события, которая в образце сливается с понятием издателя.

- С единственным общецелевым классом *PUBLISHER* подписчик может регистрироваться только у одного издателя, а у этого издателя он может зарегистрировать только одно действие, представленное *handle*; как следствие, он может подписаться только на один тип события. Это серьезное ограничение. Компонент приложения должен быть способным регистрировать различные операции у различных издателей. С этой проблемой можно было бы справиться, добавляя в методы *publish* и *handle* аргумент, представляющий издателя, так, чтобы подписчики могли выбирать издателя. Но такое решение губительно с позиций модульного проектирования, так как теперь процедуры обработки должны будут знать обо всех событиях. Еще один возможный прием – иметь несколько независимых издательских классов, по одному на каждый тип события. Это решает проблему, но в жертву приносится повторное использование.
- Поскольку издательские классы и классы подписчиков должны наследоваться от *PUBLISHER* и *SUBSCRIBER*, то непросто связать существующую модель с новым обликом без добавления существенного слоя склеивающего кода. В частности, невозможно непосредственное повторное использование существующей процедуры из модели (*op* в нашем примере) как действие, регистрируемое подписчиком. Причина в том, что в реализации *handle* эта процедура должна вызываться с аргументами, заданными издателем.
- Предыдущая проблема усугубляется в языках без множественного наследования. Издательские классы и классы подписчики наследуют от классов *PUBLISHER* и *SUBSCRIBER*, реализуя отложенные методы родителей соответственно *publish* с его фундаментальным алгоритмом и *subscribe*. Но этим классам в соответствии с их ролью в модели могут требоваться и другие родители. Единственное решение – писать больше склеивающего кода и делать эти классы клиентами соответствующих классов модели.
- Наконец, заметьте, что классы, приведенные выше, корректны по отношению к некоторой проблеме, возникающей при приводимой в литературе стандартной реализации образца. Например, обычная презентация образца «Наблюдатель» связывает подписчика и издателя в момент создания, используя издателя как аргумент процедуры создания подписчика. Вместо этого в приведенной выше реализации предусмотрена специальная процедура подписки *subscribe* в классе *SUBSCRIBER*, позволяющая связать наблюдателя с издателем в любой момент по желанию. Более того, можно отсоединиться от издателя, а позже снова с ним соединиться.
- Все эти проблемы не мешают проектировщикам успешно использовать образец в течение многих лет, но приводят к двум серьезным последствиям. Во-первых, в строящихся решениях отсутствует нужная гибкость, что является причиной дополнительной работы, например, написания склеивающего кода, присутствие связей между элементами ПО, в которой нет необходимости, что всегда плохо с позиций эволюции системы. Во-вторых, отсутствует повторное использование, так что каждый программист должен строить реализацию образца в своих интересах.

Приведенная оценка архитектурного решения образца «Наблюдатель» может служить примером, полезным и при анализе других систем. Критерии всегда одни и те же: надежность (уменьшение возможности появления «жучков»), повторное использование (минимизация усилий по интегрированию решения в новую разработку), расширяемость (минимизация усилий по адаптации приложения при добавлении новых возможностей) и простота.

18.5. Использование агентов: библиотека EVENT

Теперь мы собираемся показать, как, придав понятию типа события его полную роль, можно получить решение, устраняющее все упомянутые ограничения. Решение не только более гибкое, чем то, что мы видели до сих пор, – оно полностью повторно используемое (через библиотеку классов, которую можно использовать в качестве единственной основы API); более того, оно намного проще. Ключом являются механизмы агентов и кортежей.

Базисный API

Сосредоточимся на основной абстракции данных, следующей из обсуждения в начале этой главы, – типе события. У нас больше не будет двух замечательных классов *PUBLISHER* и *SUBSCRIBER* – да-да, единственный класс решает всю проблему, класс, называемый *EVENT_TYPE*.

В основе: два компонента характеризуют тип события.

- Подписка: объект «Подписчик» может зарегистрировать свой интерес к типу события, задав при подписке специфицируемое им действие, представленное агентом.
- Публикация: включение события

Будем использовать механизмы языка, чтобы избавиться от наиболее деликатных проблем, идентифицируемых в последнем разделе.

- Каждый тип события имеет собственную сигнатуру. Мы можем определить сигнатуру как тип кортежа и использовать его как родовой параметр универсального класса *EVENT_TYPE*.
- Каждая подписка должна задать специфическое действие. Передадим это действие как агента. Это позволит нам, в частности, повторно использовать существующий компонент бизнес модели.

Этих наблюдений достаточно для определения интерфейса класса:

```
note
       what: " Типы событий, позволяющие публикацию и подписку "
class EVENT_TYPE [ARGUMENTS -> TUPLE] feature
       publish (args: ARGUMENTS)
              – Включение события этого типа.
       subscribe (action: PROCEDURE [ANY, ARGUMENTS])
              – Регистрация действия, которое должно быть выполнено
              – для события этого типа.
       unsubscribe (action: PROCEDURE [ANY, ARGUMENTS])
              – Отмена регистрации действия (подписки) на события этого типа.
end
```

Если вы разработчик приложения, которому необходимо интегрировать схему управления событиями в систему, то вышеприведенный интерфейс (для класса, доступного в библиотеке Event) – это все, что необходимо знать.

Конечно, исследованием реализации займемся тоже, я уверен, что вы хотите увидеть ее (более интересно попытаться сначала самому ее спроектировать). Но на данный момент давайте посмотрим, как типичный клиент-программист, зная только вышеприведенный интерфейс, будет выстраивать архитектуру управления событиями.

Использование типов событий

Первое, что нужно сделать, – это определить тип события. Это просто означает задание экземпляра вышеприведенного библиотечного класса с подходящими фактическими родовыми параметрами. Например, можно определить:

```
left_click: EVENT_TYPE [TUPLE [x: INTEGER; y: INTEGER]                    [1]
– Тип события, представляющий события "щелчок левой кнопки мыши"
    once
        create Result
    end
```

Функция *left_click* возвращает объект, представляющий желаемый тип события.

Помните, что нам не требуется иметь отдельный объект для каждого события, это означало бы пустую трату пространства. Нам нужен только один объект на все события «левый щелчок». Поскольку этот объект должен быть доступен нескольким частям ПО – издателям и подписчикам, – системе требуется только один экземпляр. Эту возможность обеспечивает использование *once*-метода – метода, выполняемого только один раз,(один из немногих механизмов Eiffel, с которым мы еще не сталкивались в этой книге). Как следует из его имени, метод, помеченный как «once», вместо **do** или **deferred**, выполняет тело метода один раз при первом вызове, если таковой существует. Последующие вызовы процедуры не выполняются, а вызовы функций, как в данном случае, каждый раз будут возвращать один и тот же объект, созданный при первом вызове. Одно из преимуществ: не требуется беспокоиться о том, когда создать объект, – любая часть системы, в которой потребовалось первый раз выполнить «левый щелчок», создаст объект.

Вскоре мы увидим, где должно появиться объявление типа события **[1]**. Пока же будем полагать, что классам подписчиков и издательским классам этот объект доступен.

Чтобы включить событие, издатель – например, элемент GUI-библиотеки, который обнаруживает щелчок мыши, – просто вызывает *publish* на этом объекте с подходящим кортежем аргументов, как в нашем примере:

```
left_click.publish ([your_x, your_y])
```

На стороне подписчика также все просто – чтобы подписать действие, представленное процедурой *p* (*x*, *y*: INTEGER), достаточно вызвать на этом объекте метод *subscribe*:

```
left_click.subscribe (agent p)                                            [2]
```

Эта схема имеет замечательную гибкость, достигаемую, в частности, благодаря ответу на ожидающий решение вопрос, где объявлять тип события.

- Если нужно иметь единственный тип события, публикуемый для всех потенциальных подписчиков, просто сделайте его доступным как для издателя, так и для всех классов подписчиков, поместив объявление **[1]** в «обслуживающий» класс, к которому они все имеют доступ, например, наследуя от него.
- Заметьте, однако, что тип события – это просто обычный объект, и соответствующие компоненты, такие как *left_click*, это обычные компоненты, которые могут принадлежать любому классу. Так что издательские классы – например, классы, представляющие графические штучки, такие как *BUTTON*, – в библиотеке, такой как EiffelVision,

могут объявлять *left_click* как один из своих компонентов. Тогда схема для типичного вызова подписки приобретает вид:

```
your_button.left_click.subscribe (agent p)                    [3]
```

Это позволяет подписчику наблюдать за событиями мыши от одной вполне конкретной кнопки GUI. Такая схема реализует введенное ранее понятие контекста, здесь контекстом является кнопка.

Когда контекст является значимым, тогда подписчики подписываются не просто на тип события, как в **[2]**, а на тип, встречающийся в определенном контексте как в **[3]**. Подходящим архитектурным решением является объявление релевантных типов событий в соответствующих контекстных классах. Объявление *left_click* [1] становится частью класса *BUTTON*. Оно остается однократной (once) функцией, так как тип события – общий для всех кнопок этого вида. Объект, представляющий тип события, будет создан при первом вызове *subscribe* или *publish*. Если «левый щелчок» релевантен нескольким видам графических элементов – кнопкам, окнам, пунктам меню, – то каждый из соответствующих классов будет иметь атрибут, такой как *left_click*, одного и того же типа. Механизм «once» гарантирует, что существовать будет только один объект для типа события, более точно – один для каждого типа графических элементов.

Итак, мы получили подходящую гибкость и можем полагать, что мы справились с соответствующим пунктом (должна быть возможность делать события зависящими или не зависящими от контекста) списка требований к архитектуре «публиковать-подписываться».

- Для событий, не зависящих от контекстной информации, объявляйте тип события в общедоступном классе.
- Если контекст необходим, объявляйте тип события как компонент классов, представляющих контексты приложения. Тогда в момент выполнения он будет доступен как свойство специфического контекстного объекта.

В первом случае тип события будет иметь один экземпляр, разделяемый всеми подписчиками. Во втором – будет самое большое по одному объекту на каждый контекстный тип, для которого тип события имеет место.

Реализация типа события

Вернемся к рассмотрению сути – знакомству с реализацией *EVENT_TYPE*. Она подобна приводимой выше реализации *PUBLISHER*. Закрытый компонент subscribers хранит список подписчиков. Его сигнатура теперь такова:

```
subscribers: LINKED_LIST [PROCEDURE [ANY, ARGUMENTS]]
```

Здесь, как и прежде, *LINKED_LIST* – наивная структура, но вполне достаточная для нашего обсуждения (лучшую структуру можно увидеть в тексте фактического класса *EVENT_TYPE* из Event-библиотеки; можно также выполнить упражнение в конце главы).

Элементы, хранимые в списке, больше не являются подписчиками, понятие, в котором теперь архитектура решения не нуждается, – это просто агенты. Тип каждого такого агента, *PROCEDURE* [*ANY*, *ARGUMENTS*] указывает, что агент представляет процедуру с аргументами кортежного типа *ARGUMENTS*, как это определено в классе. Это значительно улучшает безопасность типов решения, в сравнении с тем, что мы видели ранее: несоответствия будут захвачены во время компиляции как плохие аргументы *subscribe*.

Метод *subscribe* (в наивной реализации) может выглядеть так:

```
subscribe (action: PROCEDURE [ANY, ARGUMENTS])
            — Зарегистрировать действие, которое должно быть выполнено для
            — событий этого типа.
      do
            subscribers.extend (action)
      ensure
            present: subscribers.has (action)
      end
```

Использование *ARGUMENTS* — второго родового параметра класса в типе *PROCEDURE* — гарантирует, что все процедуры будут отвергнуты, если они не имеют аргументов нужного типа.

Для публикации события мы проходим по списку и вызываем соответствующих агентов. Фактически, это тот же код, что и в классе *PUBLISHER* в образце «Наблюдатель», хотя *args* теперь имеют более подходящий тип — *ARGUMENTS*:

```
publish (args: ARGUMENTS)
            — Публиковать событие подписчикам.
    do
          — Включить событие этого типа.
          from subscribers.start until subscribers.after loop
             subscribers.item.call (args)
             subscribers.forth
          end
end
```

Любой аргумент при вызове метода агента должен быть кортежем, поскольку *ARGUMENTS* ограничены типом кортежа.

Только что описанное решение лежит в основе библиотек Event и EiffelVision GUI. Оно широко используется для графических приложений, как небольших, так и сложных, включая само окружение EiffelStudio.

Класс включает еще несколько деталей, которые стоит внимательно рассмотреть.

Время чтения программ!

> **Типы событий**
> Познакомьтесь с текстом библиотечного класса *EVENT_TYPE* и убедитесь, что вы все в нем понимаете.

18.6. Дисциплина подписчиков

Если применять любой из приемов этой главы — от несовершенного образца «Наблюдатель» до механизма, основанного на агентах, — то следует уделить внимание проблемам производительности, которые могут приводить к потенциально разрушительным потерям памяти. Избежать их достаточно просто, если корректно определять поведение подписчиков.

Почувствуй методологию

Не забывайте вовремя отменять подписку

Если вы знаете, что после некоторого этапа выполнения системы некоторых подписчиков уже не нужно уведомлять о происходящих событиях определенного типа событий, то не забудьте включить подходящий вызов *unsubscribe*.

Чем вызвано это правило? Проблемой в использовании памяти. Из реализации *subscribe*, как для версии *PUBLISHER* в образце «Наблюдатель», так и для версии из класса *EVENT_TYPE*, следует, что издатель записывает в список подписчиков ссылку на объект подписчика. В приложениях GUI издатель принадлежит облику, а подписчик – модели. Поэтому объект облика сохраняет ссылку на объект модели, который, в свою очередь, может содержать другие ссылки на многие объекты модели (например, самолеты, полеты, расписание и так далее в системе управления полетами).

Но тогда становится невозможным, если только сами объекты облика становятся бесполезными, избавиться от такого модельного объекта, даже если он вычислению уже не нужен. В современных средах разработки сборщик мусора никогда не будет собирать объекты, на которые ссылаются другие объекты. Если же освобождение памяти выполняется вручную (как в C и C++ окружении), то ситуация становится еще хуже. В любом случае, появляется источник «потерь памяти»: выполнение не справляется с возвращением неиспользуемых объектов, объем занимаемой памяти продолжает расти.

Отсюда следует вышеприведенное правило: подписка – это прекрасно, но как только сервис вам больше не нужен, не забывайте прекратить подписку, как это вы делаете с удалением рекламных журналов и каталогов, чтобы они не заполняли ваш почтовый ящик.

Методологические правила никогда не бывают столь же эффективными, как инструментарий и архитектура, гарантирующая желаемые цели. В данном случае нет очевидного способа автоматической ликвидации ненужной подписки – остается полагаться на методологические советы.

Когда вы подписываетесь на агента и хотите позже иметь возможность отмены подписки, следует использовать переменную, представляющую агента:

```
handler:=agent p                                                              [4]
left_click.subscribe (handler)
            … – Тогда, когда пришло время отмены подписки:
left_click.unsubscribe (handler)
```

Подписка через переменную, задающую агента, вместо его непосредственного использования, как в предыдущих примерах *left_click.subscribe* (**agent** *p*), гарантирует, что отмена подписки применяется к тому же самому объекту (в отличие от *left_click.unsubscribe* (**agent** *p*), который мог быть применен к новому объекту). Если вы уже подписали данный обработчик более одного раза к данному типу события, отмена (использующая *remove_all*) удалит все такие подписки.

18.7. Архитектура ПО. Уроки

Проекты, рассмотренные в этой главе, позволяют сделать некоторые общие выводы об архитектуре ПО.

Выбирайте правильные абстракции

Наиболее важным моментом в проектировании ПО, по крайней мере, для ОО-подхода, является идентификация правильных классов – абстракций данных (на следующем месте по важности – идентификация отношений между этими классами).

В образце «Наблюдатель» ключевыми абстракциями являются «Издатель» и «Подписчик». Обе концепции полезны, но приводят к созданию несовершенной архитектуры. Базисная причина в том, что отсутствует достаточно хорошая абстракция для парадигмы «публиковать-подписать». С первого взгляда все кажется приемлемым еще и потому, что совместно оба слова удачно характеризуют стоящую проблему. Однако хорошую абстракцию характеризуют не привлекательные имена, а множество согласованных компонентов. Единственной важной компонентой издателя является то, что он публикует события данного типа, а единственно важной компонентой подписчика является возможность подписаться на события данного типа. Этого не хватает для полного освещения проблемы.

Наиболее важной концепцией, не обнаруживаемой образцом «Наблюдатель», является понятие типа события. Эта ясно осознаваемая абстракция данных с несколькими характеристическими компонентами: командами для публикации и подписки на события, понятием аргумента (которое имеет даже большее значение, чем сеттер-команды и запросы).

Рассматривая EVENT_TYPE как ключевую концепцию, приходим к основному классу финального проектирования. Тем самым удается избежать наследования от родителей специального вида для классов подписчиков и издательских классов. Издатель теперь – это простой элемент ПО, использующий *publish* метод для некоторого типа события, а подписчик – элемент, применяющий метод *subscribe*.

Вернемся к МОК

Одним из следствий последнего проекта является упрощение общей архитектуры, предполагаемой парадигмой МОК («модель – облик – контроллер»). Контроллер – это часть, являющаяся «склеивающим кодом», и мы должны стараться свести ее к минимуму.

Сердцем архитектуры контроллера является класс *EVENT_TYPE*. В простейшей схеме этого достаточно, если позволять элементам модели непосредственно подписываться на события:

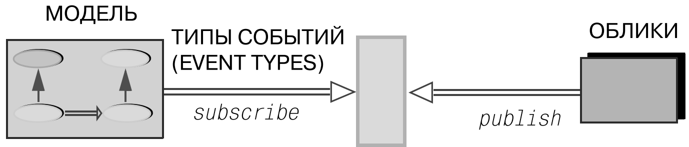


Рис. 18.6. Прямая подписка

(Двойная стрелка, как обычно, означает клиентское отношение, используемое здесь для реализации более абстрактных отношений общей МОК-картины)

На этой схеме нет явных компонентов контроллера.

Пока модель не знает непосредственно об обликах (если не используется контекст), она связывается со специфическими типами событий. Такой подход имеет как ограничения, так и преимущества.

- Негативная сторона в том, что изменение обликов становится затруднительным. Если не ограничиваться одним обликом, то новые облики должны включать одни и те же со-

бытия. Это предполагает, что различные облики не полностью различны, например, один из них может задавать интерфейс GUI, а другой – Web.

- С другой стороны, преимущество в большой простоте. Элементы модели могут непосредственно указать, какие действия следует выполнять для специфических событий, приходящих от интерфейса. Это позволяет не иметь склеивающего кода.

Эта схема хороша для относительно простых программ, где интерфейс или, по меньшей мере, стиль интерфейса, известен и стабилен. Для более сложных случаев можно ввести явный контроллер, в задачу которого входит отделение от модели подписки на тип события.

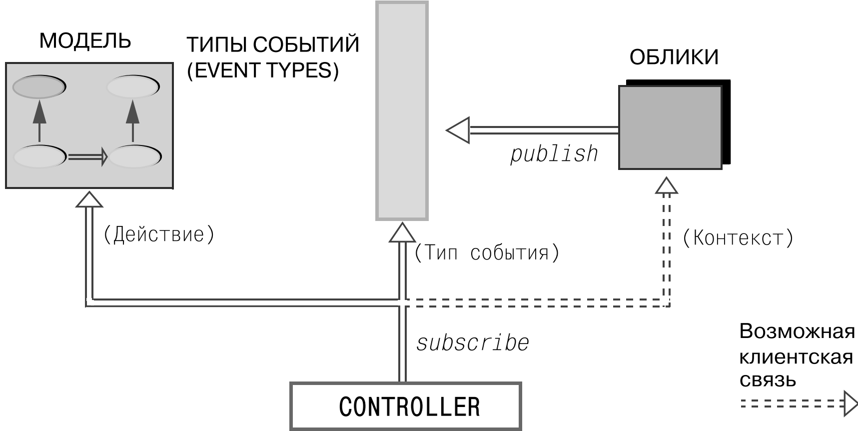


Рис. 18.7. Подписка через контроллер

Контроллер теперь становится независимой частью ПО. Его работа состоит в подписке элементов модели на типы события. Он будет иметь связи с:

- моделью, так как аргументы *subscribe* являются действиями, задаваемыми при подписке, и здесь должны быть агенты, встроенные в механизмы модели;
- обликами, если используются контексты. На рисунке это отображено возможной клиентской связью.

Это решение приводит к полному расщеплению модели и облика. В типичном приложении контроллер – как правило, небольшой компонент, достигающий цели и использующий склеивающий код в минимальной степени, насколько это возможно.

Модель как издатель

В рассматриваемых до сих пор схемах GUI все события приходят обычно через механизм библиотек и обрабатываются моделью. Другими словами, издателями являются облики, а элементы модели – подписчиками.

Схему можно расширить, позволив модели играть роль издателя. Например, если элемент GUI, такой как «сектор круговой диаграммы», отражает множество значений, которое может изменяться в модели, то модель становится причиной события «change». Облики становятся подписчиками этого типа события.

Эта возможность просто добавляется во второй схеме – подписке через контроллер. Контроллер будет действовать полностью как двусторонний брокер, получающий события от обликов для их обработки в модели и аналогично работающий в обратном направлении.

Это решение, добавляющее сложность контроллеру, полезно только при наличии нескольких обликов.

Утром инвестиции, вечером прибыль

Общим для двух архитектур – «Наблюдатель» и «Библиотека Event» – является необходимость предварительной подписки на тип события до начала их обработки.

У подписчиков есть возможность подписки и ее отмены в любое время. Для Event-решения можно создавать новые типы событий на любом этапе вычислений. Такая гибкость может быть полезной, но в типичном сценарии четко выделяются два этапа выполнения.

- На этапе инициализации подписчики регистрируют свои действия, типично приходящие из модели.
- Затем начинается этап, соответствующий бизнес-решению задачи. На этом этапе структура управления становится структурой, управляемой событиями. Выполнение отслеживает включение событий издателями, что (возможно, в зависимости от контекста) становится причиной запуска действий, предписанных моделью.

В соответствии с таким порядком правильнее было бы называть парадигму нашего образца «Подписаться-Публиковать». Это напоминает успешного инвестора: вначале он инвестирует в новое дело, а уж потом, когда оно начинает работать, может получить прибыль.

Возможно, вы вспомнили вариант этого общего подхода – стратегию компиляции, применяемую при топологической сортировке: сперва данные транслируются в подходящую структуру, затем начинается их обработка.

Оценка архитектуры ПО

Ключом качества программных систем является их архитектура, покрывающая такие аспекты, как:

- выбор классов, основанных на подходящей абстракции данных;
- решение о связях между классами. Целью является минимизация таких связей (при сохранении возможности независимой модификации и повторного использования различных частей ПО);
- для связанных классов – выбор правильного отношения, клиентского или наследования;
- задание компонентов классов;
- задание для классов и их компонентов правильных контрактов;
- при задании контрактов – выбор между «требовательным» стилем (сильные предусловия, заставляющие клиентов поставлять проверенные значения исходных данных), толерантным стилем (противоположный подход) или применением промежуточного подхода, возлагая часть проверок на клиента, часть – на соответствующий метод;
- удаление ненужных элементов;
- недопущение дублирования кода и удаление его, если оно уже присутствует. Для этих целей применяйте наследование: если есть общие части у двух и более классов, то сделайте эти классы наследниками общего родителя, который и будет хранителем общности, присущей нескольким классам. Для устранения дублирования используйте также универсальность, кортежи и агентов;
- использование преимуществ известных образцов проектирования;
- проектирование инструментария, пригодного для повторного использования: простого, легкого для понимания, поставляемого с подходящими контрактами;
- обеспечение согласованности: в программной системе подобные цели должны достигаться подобными методами. Это общее требование покрывает все вышеупомянутые

требования, например, если для некоторых классов выбрано отношение наследования, то для схожей пары классов также должно выбираться наследование, а не клиентское отношение. Согласованность важна и при построении инструментария – API, если программист изучил некоторую группу классов, то он должен ожидать, что подобные соглашения будут действовать и для других классов.

Эти же задачи возникают и при улучшении существующих проектов – деятельность, известная как рефакторинг. Хорошей идей является критический обзор существующей программной системы. Но помните, что лучше не делать ошибки, чем их исправлять. Проектируйте *хорошо* с самого начала.

Вне зависимости от того, идет ли речь о начальном проектировании или о рефакторинге, работа над архитектурой системы сложна и в то же время полезна. Обсуждение в этой главе и некоторых других (глава о топологической сортировке) дают некоторые идеи того, что нужно делать. Критерии успеха всегда одни и те же.

Почувствуй методологию

Оценка архитектуры ПО

Критически обсуждайте альтернативы при анализе возможных решений проектирования. Критериями всегда являются: надежность, расширяемость, повторное использование и простота.

18.8. Дальнейшее чтение

Рис. 18.8. Трюгве Реенскауг

Рис. 18.9. Гамма (2007)

Trygve Reenskaug, heim.ifi.uio.no/~trygver/themes/mvc/mvc-index.html.

Статьи по МОК.

Трюгве Реенскауг, известный норвежский ученый в области информатики. Работая в 1979 году в Xerox PARC (известный Исследовательский центр в Пало-Альто), ввел образец МОК. Приведенная ссылка позволяет познакомиться с его работами по этой теме. Я полагаю, что оригинальная работа по МОК 1979 года все еще остается лучшей презентацией МОК.

Erich Gamma, Richard Helm, Ralph Johnson and John Vlissides: Design Patterns, Addison-Wesley, 1994.

Классический текст по образцам проектирования. Содержит среди многих других образцов стандартное описание образца «Наблюдатель».

На русском языке: Эрих Гамма, Ричард Хелм, Ральф Джонсон, Джон Влиссидис: «Приемы Объектно-ориентированного проектирования. Паттерны проектирования», Питер, 2006 г.

Bertrand Meyer: The Power of Abstraction, Reuse and Simplicity: An Object-Oriented Library for Event-Driven Design, in From Object-Orientation to Formal Methods: Essays in Memory of Ole-Johan Dahl, eds. Olaf Owe, Stein Krogdahl, Tom Lyche, Lecture Notes in Computer Science 2635, Springer-Verlag, 2004, pages 236-271.

se.ethz.ch/~meyer/publications/lncs/events.pdf.

Важная часть этой главы построена на материалах этой работы, в которой анализируются три решения: образец «Наблюдатель», механизм делегатов .NET и библиотека Event.

18.9. Ключевые концепции, изученные в этой главе

- Проектирование, управляемое событиями, называемое также концепцией «публиковать-подписаться», приводит к системам, чье выполнение управляется откликами на события в противоположность традиционным управляющим структурам. События включаются в программной системе, часто в ответ на внешние события. Графический интерфейс – GUI-программирование – характеризует важную область приложений.
- Ключевой абстракцией проектирования, управляемого событиями, является понятие типа события.
- Издателями являются элементы программной системы, которые могут включать определенные типы событий. Подписчиками являются элементы программной системы, которые регистрируют действия, выполняемые при возникновении событий определенного типа. При возникновении события подписчики получают уведомление, выполняя в ответ предписанное действие.
- В системе с одним или более *обликов* важным правилом проектирования является отделение обликов от ядра приложения, называемого *моделью*.
- Архитектура МОК (MVC – Model – View – Controller) предполагает существование контроллера – промежуточного слоя между моделью и обликом, управляющего взаимодействием с пользователями.
- Образец «Наблюдатель» предлагает решение проблемы за счет введения двух классов высокого уровня *PUBLISHER* и *SUBSCRIBER*, от которых наследуются издательские классы и классы подписчиков. Каждый класс подписчик задает процедуру, которая описывает действие, выполняемое в ответ на событие. Каждый издательский объект имеет закрытое от клиентов свойство, хранящее список его подписчиков. Благодаря динамическому связыванию, при включении события выполняются желаемые действия, специфические для каждого подписчика.
- Агенты, ограниченная универсальность и кортежи позволяют дать общее решение проблемы проектирования, управляемого событиями, на основе единого повторно используемого класса, основанного на центральной абстракции: *EVENT_TYPE*.
- Архитектура ПО – ключ к его качеству. Проектирование архитектуры новой системы и улучшение существующей (рефакторинг) постоянно должно быть в центре внимания, фокусируясь на простоте, надежности, расширяемости и повторном использовании.

Новый словарь

Application domain	Проблемная область приложения	**Argument (of an event)**	Аргумент (события)
Business model	Бизнес-модель	**Catching (an event)**	Захват (события)
Context (of an event)	Контекст (события)	**Control (Windows)**	Элемент управления
Controller	Контроллер	**Event**	Событие
Event-driven	Управляемое событиями	**Event type**	Тип события
		External event	Внешнее событие
Glue code	Склеивающий код	**Handle (an event)**	Обработка (события)
Model	Модель	**MVC**	МОК
Publish (an event)	Публикация (события)	**Publish-Subscribe**	Публиковать-подписаться
Register	Регистрация		
Refactoring	Рефакторинг	**Signature (of event type)**	Сигнатура (типа события)
Subscribe	Подписаться	**Trigger (an event)**	Включение (события)
View	Облик	**Widget**	Виджет, элемент управления

18-У. Упражнения

18-У.1. Словарь

Дайте точное определение терминов словаря.

18-У.2. Карта концепций

Добавьте новые термины в карту концепций, построенную в предыдущих главах

18-У.3. Эффективный «Наблюдатель»

Выберите подходящее представление для списка подписчиков, адаптируя реализацию образца «Наблюдатель» так, чтобы все следующие операции выполнялись за время O(1): добавить подписчика (ничего не делать, если таковой уже существует в списке); удалить подписчика (ничего не делать, если такового нет в списке), выяснить, занесен ли потенциальный подписчик уже в список. Процедура публикации *publish*, если не учитывать время, затрачиваемое на реальное выполнение действия по обработке события, должна работать O(*count*) времени, где *count* – число подписчиков, реально подписанных на данный тип события. Структура данных, содержащая подписчиков, должна быть разумной и по памяти – O(*count*) (*подсказка*: рассмотрите различные структуры данных в главе 13 и соответствующие классы EiffelBase). Заметьте, что эта оптимизация также применима и к реализации классов из библиотеки Event.

18-У.4. Безопасный по типам «Наблюдатель»

Покажите, что при реализации образца «Наблюдатель» возможна схема типов, лишенная недостатков как схемы аргументов 1, так и схемы аргументов 2. Такая схема использует возможности, показанные в этой главе и реализованные в библиотеке Event: ограниченную универсальность и кортежи. Ваше решение должно как описать изменения в классах *PUBLISHER* и *SUBSCRIBER,* так и представить наследуемые от них типичные классы издателя и подписчика.

Часть V

Цель – инженерия программ

Эта часть содержит заключительную главу, предшествующую приложениям. В ней рассматриваются требования, необходимые для перехода на новый уровень, перехода от простого программирования к профессиональной разработке ПО промышленного качества. Этот уровень называется программной инженерией.

19

Введение в инженерию программ

Разработка ПО – это нечто большее, чем программирование. Это утверждение является не парадоксом, а осознанием всех факторов, влияющих на успех программного проекта, и всех возникающих задач, не связанных с написанием программ, но требующих внимания. Вот несколько примеров.

- Блестящая программа может быть признана неудачной, если ее интерфейс не воспринимается целевой аудиторией.
- Прекрасная программа бесполезна, если она не решает нужную задачу. Следовательно, необходимо описание требований к системе, описывающих ее точную функциональность и охватывающих потребности конечных пользователей.
- Помимо технических аспектов проекты должны принимать во внимание проблемы управления: установление и выдерживание контрольных сроков, организация встреч и других видов коммуникации между членами команды, реализующей проект, определение бюджета и управление расходами.

Эти и другие виды деятельности, обсуждаемые в этой главе, не включают приемы программирования, но если не принимать их во внимание, они могут разрушить проект, независимо от его технического совершенства.

В предыдущих главах мы почти исключительно сосредоточивались на программировании, но картина была бы не полна без рассмотрения непрограммистских аспектов инженерии программ.

Это широко разветвленная и хорошо разработанная дисциплина. Для ее полного рассмотрения требуется отдельный учебник. К счастью, несколько учебников уже существует, ссылки на них можно будет найти в конце этой главы в разделе «Дальнейшее чтение». Данная глава имеет более ограниченные цели: ввести несколько непрограммистских аспектов программной инженерии, достаточных, чтобы пробудить ваш интерес и желание прочесть имеющуюся литературу по этой теме.

Важную роль в инженерии программ играют инструментальные средства, применяемые при разработке. Обзор некоторого инструментария появлялся в предыдущих главах (глава 12).

19.1. Базисные определения

Следующее определение, охватывающее широкую область, будет нам полезно.

Определение: инженерия программ

> Инженерия программ – это множество технических приемов, включающих теорию, методы, процессы, инструментарий и языки – для разработки и оперирования с программным продуктом, отвечающим определенным стандартам качества.

Два важных свойства инженерии программ устанавливаются в этом определении – ПО должно быть программным продуктом и иметь определенное качество.

Программный продукт – это ПО, предназначенное для функционирования в реальном окружении и для решения реальных задач. Разрабатываемое в экспериментальных целях ПО – «программа на выброс», используемая однократно, не сопровождаемая в течение длительного времени, – не квалифицируется как продукт. Исключением будет случай, когда эксперимент является частью достижения широкой цели, принадлежащей инженерии программ. Например, эксперимент по оценке различных возможных алгоритмов сам по себе не квалифицируется, но все меняется, если он выполняется как часть разработки программного продукта.

Программный продукт должен удовлетворять комбинации ограничений. Они могут включать:

- ограничения качества (обсуждаемые далее): например, гарантию, что система будет функционировать без отказов, давать корректные результаты и быстро работать;
- ограничения размера: программный продукт может состоять из тысяч или десятков тысяч классов и других модулей и сотен тысяч или миллионов строк кода;
- временную протяженность: системы, используемые в индустрии, часто должны сопровождаться (постоянно использоваться и регулярно обновляться) в течение многих лет и даже десятилетий;
- командную разработку: такие системы могут включать большие команды разработчиков и огромное число пользователей. Это приводит к появлению проблем с коммуникацией и управлением разработкой;
- ограничения влияния: эти системы воздействуют на физические процессы и процессы, влияющие на жизнь людей, – поезда могут не приходить во время, телефоны не работать, зарплата не выплачиваться, заказы не выполняться, а фактически все может быть гораздо хуже. Этим объясняются и требования к качеству.

Забота о качестве находится в центре инженерии программ. Определение говорит об «определенных стандартах» качества. Качество не есть нечто такое, что кто-то произвольно объявил присутствующим или отсутствующим в программном процессе или продукте, оно должно оцениваться настолько объективно, насколько это возможно.

Определение также говорит о «разработке и оперировании». Программная конструкция не возникает мгновенно, наряду с разработкой появляется процесс фактического использования (оперирования) ПО. Даже фаза разработки не должна пониматься как создание некоторого законченного выпуска системы: то, что происходит после этого, также имеет значение. Мы уже встречались с техническим термином для этого вида деятельности.

Определение: сопровождение ПО

> **Сопровождение** программной системы включает все виды деятельности, встречающиеся после выпуска первой версии работающей системы: адаптация новых платформ и окружений, коррекция замеченных недостатков, расширение (добавление новой функциональности), удаление избыточной функциональности, улучшение качества.

Термин «сопровождение» пришел из других областей инженерной деятельности – сопровождение машины или дома. Как указывают многие, аналогия «хромает», поскольку программа не повреждается от частого использования в отличие от предметов материального мира – после тысячи запусков она остается такой же, как и при первом запуске. Однако как программистский термин «сопровождение» продолжает использоваться – и никаких проблем не возникает, если понимать его в соответствии со сделанным определением.

Жаргонный термин будет полезен для обсуждения.

Сопричастник[1]:

> Сопричастником программного проекта является любой человек, который воздействует на проект и на его качество или который оказывается под влиянием проекта с достигнутым качеством.

Понятие охватывает множество людей: разработчиков, тестеров, менеджеров проекта, других членов команды. Но сюда же включаются и будущие пользователи системы, все те, на кого проект воздействует, в их числе и менее приятная, но все же существующая часть людей – те, кто не станет пользователями, поскольку система сделает их текущую работу бесполезной. К сопричастникам относятся маркетологи и люди, занимающиеся продажей, в чьи обязанности входит поиск потребителей программного продукта после его выпуска, тренеры, обучающие работе с продуктом, ИТ-отделы в корпорациях, приобретающих продукт.

Важной частью управления проектом является идентификация всех сопричастников на ранних этапах, чтобы рассмотреть их потребности и ограничения каждой из групп.

19.2. DIAMO – облик инженерии программ

Для понимания вызовов, стоящих перед инженерией программ, можно рассматривать эту дисциплину как состоящую из пяти частей – пяти граней равной важности. Программирование – это первый компонент одной из частей (второй – реализация). В качестве мнемоники этой классификации будем использовать акроним DIAMO (по первым буквам английских слов, именующих части дисциплины – Describe, Implement, Assess, Manage, Operate. Акроним является префиксом слова *Diamo*nd – бриллиант).

Описание (Describe): многие виды деятельности в инженерии программ включают понимание и специфицирование задач и систем. Целью такой деятельности является не построение решения, а описание свойств решения. Описание может требоваться перед построением решения на этапе анализа требований и проектирования спецификаций или на более поздних этапах при создании документации

[1] Термин, используемый в оригинале – stakeholder. Мне не известен его хороший перевод. Пришлось использовать новое слово, которое, как мне кажется, точно передает смысл понятия.

Рис. 19.1. DIAMO. Бриллиант инженерии программ

Реализация(Implement): задача построения программ. Этап включает не просто реализацию (программирование в узком смысле этого термина), но и проектирование – определение архитектуры системы на высоком уровне абстракции.

Оценка(Assess): большая часть процесса связана с анализом ПО. Продукты, подлежащие оценке, – это не только программы, но и все, что входит в понятие ПО, в частности, архитектура проекта и документация. Наиболее общей целью является выявление погрешностей (можно сказать по-другому – установление корректности). На этом этапе решаются такие важные задачи, как статический анализ программ, тестирование и отладка. Это, однако, далеко не все, что требуется оценить. В частности, при эффективной организации процесса разработки требуются количественные показатели как продукта (размер, сложность, ...), так и процессов (время, стоимость), что и является предметом соответствующих метрик.

Управление (Manage): любой серьезный проект, даже с несколькими разработчиками, требует управления. К этой части относятся такие задачи, как установление коммуникаций в команде разработчиков. Сегодня проблема усложняется из-за того, что многие современные проекты выполняются географически распределенной командой, на это накладываются и языковые барьеры. Сюда же относятся задачи составления расписания, выдерживания контрольных сроков, установление взаимодействия с пользователями и другими сопричастниками, управление неизбежными запросами на изменение.

Функционирование(Operate): когда все проанализировано, спроектировано, реализовано, протестировано и документировано, тогда только все и начинается – система должна функционировать. Она должна быть развернута и начать успешно работать. Фаза развертывания в индустрии может быть главным камнем преткновения. Представьте себе инсталляцию банковского ПО, которое должно работать в тысячах автоматов во многих странах, с необходимостью учета местного контекста (языка дисплея, требований безопасности, сетевых соединений). Сюда же относится обеспечение устойчивого процесса будущих обновлений системы.

За программированием, конечно же, остается его фундаментальная роль: нет программ – нет и инженерии программ. Все другие виды деятельности теоретически не обязательны. На практике любой серьезный проект должен уделять внимание всем сторонам инженерии программ.

19.3. Составляющие качества

Качество – главная цель инженерии программ – является понятием со многими различными характеристиками, часто называемых факторами качества. Давайте рассмотрим некоторые из наиболее важных факторов.

Процесс и продукт

Обсуждение инженерии программ предполагает два дополняющих друг друга аспекта.

- **Продукты**, представляющие результаты разработки. Наиболее очевидным продуктом является исходный код, но программные проекты часто включают и такие продукты, как документы требований и проектирования, тестовые данные, планы проекта, документацию, процедуры инсталляции.
- **Процессы** – механизмы, используемые для получения этих продуктов.

Число ошибок в поставляемой программе является примером проблемы продукта. Поставляется ли программа в соответствии с расписанием – пример проблемы процесса.

В каждом случае оба аспекта играют свою роль: процесс определяет, в частности, обнаружение и удаление ошибок, а выдерживание сроков влияет на продукт, например, из-за опускаемой частично функциональности.

Факторы качества ПО удобно обсуждать, рассматривая их с трех разных позиций.

- Качество процесса, характеризуемое эффективностью процесса разработки.
- Непосредственное качество продукта, характеризуемое адекватностью продукта в том виде, как он поставляется в последней своей версии.
- Долговременное качество продукта, характеризуемое будущими перспективами. В мире инженерии программ, где проекты могут проживать длинную жизнь, эти факторы столь же важны, как и непосредственная картина.

В каждой области рассмотрим главные цели, начав с наиболее очевидного свойства – качества непосредственного продукта. Обсуждение включает некоторые комментарии, объясняющие, почему некоторые другие факторы играют не столь важную роль. Два общих замечания об этом обзоре:

- Не будут даваться никакие явные определения поясняемых факторов качества («простота использования», «легкость изучения»). Соответствующие термины не появятся в разделе «Новый словарь» в конце этой главы.
- Можно заметить некоторую относительность в определениях: адекватность означает удовлетворение определенным потребностям, эффективность – это адекватное использование ресурсов. Это не приводит к неопределенности, скорее наоборот, определение целей качества ПО полезно лишь в той мере, насколько они позволяют оценивать продукты или процессы по отношению к достижению поставленных целей. Определения, следовательно, предполагают, что такие цели явно заданы.
- Эта проблема не просто академическая: вообразите, что вы руководитель проекта и отслеживаете число оставшихся погрешностей в проекте. Когда вы дадите санкцию на выпуск очередного релиза системы: при числе ошибок, равном 1000, 500, 200, 0? В реальной ситуации необходимо классифицировать погрешности: приводящие к зависаниям системы; серьезные, но не критические; небольшие проблемы, такие как несовершенство пользовательского интерфейса; пропущенная функциональность из разряда «хорошо бы иметь», которая может быть отложена на следующий выпуск. На поставленный вопрос невозможно ответить, если не иметь четких критериев, установленных заблаговременно. Мы вернулись назад к оригинальному определению инженерии программ и его требованию «отвечать определенным стандартам качества».

На следующем рисунке показана общая классификация факторов качества, которые предстоит рассмотреть.

Качество непосредственного продукта

Качество продукта включает следующие факторы.

- **Адекватность**: удовлетворение определенным потребностям пользователя. Другими словами, продукт должен служить целям сообщества пользователей. Другие факторы, часто упоминаемые в этой области, – это полнота и полезность, но оба они менее точ-

Рис. 19.2. Факторы качества ПО

ные и могут рассматриваться как частные случаи адекватности. Никогда система не обладает полной функциональностью, поскольку всегда найдется такой пользователь, чьи потребности не удовлетворяются системой. Полезность также является субъективным критерием, если только не установлены четкие критерии «полезности».

- **Корректность**: выполнение функций системы в соответствии с предписанными спецификациями в случаях, покрываемых спецификациями. Ясно, что это фундаментальное требование. Точно так же ясно, что корректность трудно достижима. Непросто написать программу, работающую в точном соответствии со спецификациями, но и написание самих спецификаций трудная задача, – необходимо учитывать все возможные случаи. Непросто и написать однозначно понимаемый документ со спецификациями.

  Важным следствием этого определения является вывод, что корректность – понятие относительное. Никогда нельзя сказать, что программная система корректна или некорректна. О корректности системы можно говорить только по отношению к заданной спецификации. В математических терминах «корректность» применяется не к программе, а к паре [программа, спецификация].

- **Устойчивость**: насколько хорошо система реагирует на ошибочные ситуации ее использования, выходящие за пределы спецификации. Пользователь мог нажать не ту кноп-

ку, сенсор мог неправильно сработать, другая программа прислала ошибочный ввод, – все такие ситуации не должны быть причиной отказа системы от работы или выдачи неверных результатов. Устойчивость предполагает обработку ошибок и механизмы восстановления системы.

- **Безопасность**: насколько хорошо система защищает себя, свои данные, своих пользователей и любые связанные с ней устройства от враждебных попыток нарушить ее работу. К несчастью, говоря об устойчивости, необходимо заботиться не просто об ошибках, – компьютерные системы часто являются целями преднамеренных вражеских атак. Поэтому нельзя писать ПО, особенно работающее в сети, не учитывающее потенциальных угроз.
- **Эффективность** (часто называемая производительностью): адекватное использование времени, памяти и других ресурсов, таких как пропускная способность для систем, осуществляющих передачу данных по сети. Мы говорим об адекватности, но не об оптимальности. Если компилируемой системе требуется 1 MB памяти, но можно уменьшить объем до 0.6 MB, то такая оптимизация вовсе не обязательно будет полезной, особенно если требует дополнительных затрат. Если вы ожидаете, что ваши пользователи будут иметь достаточно памяти, то лучше потратить время на другие факторы качества. Но если речь идет об устройствах с малой памятью, то оптимизация требуемого пространства может стать критической. И снова речь идет о необходимости объективных критериев.
- **Простота использования**: непростой выбор – сделать систему простой в использовании для различных групп пользователей. Часто систему пытаются сделать простой для новичков, но не менее важно помогать экспертам, которые точно знают, что им необходимо, и не хотят проходить через подсказки и информационные окна. Так что «простоту» необходимо поддерживать в широком диапазоне от новичков до экспертов. Каждый из нас новичок в использовании одних инструментов и эксперт – в других. Каждый из нас проходил путь от новичка к эксперту, и система должна поддерживать нас на этом пути.
- **Легкость (простота) обучения**: понятие, тесно связанное с предыдущим фактором.

Долговременное качество продукта

Некоторые качества продукта не оказывают непосредственного влияния на его текущих пользователей, но важны для тех, кто занимается его поставкой. Если я водитель автомобиля, то меня не очень волнует простота модификации ПО, отвечающего за работу кондиционера, мне важно, чтобы кондиционер хорошо работал (непосредственный фактор). Но если я отвечаю за разработку ПО для Nissan или BMW, то я должен учитывать долговременную перспективу: будет ли система просто обновляться, сможет ли версия, разработанная для седанов, быть преобразована за разумные деньги в версию для кабриолетов?

При описании ПО часто встречается термин «пользователь», приобретший уже почти мифический смысл. Хорошо заботиться о пользователях, но с точки зрения долговременной перспективы сопричастники включают и другие группы, о которых следует заботиться. Более общие термины – клиенты, заказчики – применимы как к нынешним пользователям, так и к тем, кто работал или будет работать с системой в будущем.

Качества, ориентированные на долгий период, включают:

- **простоту исправлений**: насколько просто обновить ПО для устранения погрешностей (корректности, устойчивости, безопасности, простоты использования ...). Одним из рецептов по достижению этого фактора является структура системы: необходимо проектировать модульную структуру, легко понимаемую и отражающую структуру стоящей проблемы и ее решения;

- **расширяемость**: простота добавления функциональности. И здесь ключом является структура. Изученная нами ОО-техника – абстракция данных, скрытие информации, классы, универсальность, контракты, наследование, динамическое связывание, агенты и прочее – облегчает расширение. Расширяемость является принципиальным требованием практической разработки ПО, поскольку функциональность практически каждой системы подвергается изменениям.

  Причины изменений могут быть разные: в начальном определении требований пропущены некоторые функции, отложенные на время возможности, как следствие успеха системы, ведущее к ее развитию. Хороший процесс разработки выстраивает дисциплину таких изменений, определяя строгие процедуры анализа новых запросов;

- **переносимость**: насколько просто перенести ПО на другие платформы. Под платформой здесь понимается комбинация архитектуры компьютера и его операционной системы плюс другие ресурсы, необходимые системе, такие как системы управления базами данных. За последние десятилетия в ИТ-индустрии накоплен большой опыт по конструированию переносимого ПО. Для общецелевых вычислений на рынке компьютерного железа предлагается несколько архитектур (Pentium и совместимые с ним, Sparc, PowerPC), мир операционных систем – это Windows, Unix-варианты, такие как Linux, Solaris и Mac OS. Что же касается языков программирования, то большинство из них доступны на разных платформах;
- **повторное использование**: какая доля продукта может быть использована в будущих разработках. Многим приложениям нужна одна и та же функциональность, либо общей природы (структуры данных и фундаментальные алгоритмы, механизмы GUI), либо нацеленная на одну и ту же прикладную область. *Повторно используемое ПО* – это ПО, достаточно независимое от конкретного проекта и допускающее использование в последующих проектах. Объектная технология направлена на повторное использование и ведет к построению **программных компонентов**, служащих потребностям многих разработок (подумайте о библиотеке Traffic и обо всех библиотеках, на которых она сама основана). Даже не создавая целенаправленно программных компонентов, следует прилагать усилия по созданию ПО, допускающего использование в будущих проектах.

В литературе можно найти ссылки на фактор качества, называемый **сопровождаемостью**, предполагающий простоту работы с системой после выпуска начального релиза. Это важное понятие не является независимым фактором, а представляет комбинацию уже рассмотренных долговременных факторов, так как сопровождаемость включает и фиксацию ошибок, и расширение функциональности и перенос на другие платформы.

Все рассмотренные до сих пор свойства являются внешними факторами качества: они представляют прямой интерес для клиентов. Качество включает и внутренние факторы, характеризующие то, как фактически написана система, эти факторы значимы только для разработчиков. Неформально со многими из них вы знакомы, поскольку они соответствуют советам по проектированию и программированию, проходящим через всю книгу. При проектировании ПО классы должны отражать релевантные абстракции данных, между классами должны устанавливаться подходящие отношения (наследования и клиентские), необходимо использовать преимущества разработанных образцов проектирования, включать осмысленные контракты, применять скрытие информации, встраивать в проект документацию и использовать комментарии, писать проект в читаемом стиле, облегчающем его будущее расширение.

Еще одним примером внутренних факторов является список свойств, определяемых ниже. Этими свойствами должны обладать правильно построенные документы, задающие тре-

бования к системе, но некоторые из них применимы и к программам. Внутренние качества – это тот фундамент, на котором строится качество системы. Внешние факторы, внешние свойства достижимы только при высоком внутреннем качестве системы. Корректность и простота исправлений, например, сводятся в конечном итоге к систематическому программированию, архитектурному стилю и правильно установленным контрактам.

Но нужно понимать, что, в конечном счете, значение имеют внешние факторы, так как они напрямую связаны с потребностями клиентов, в интересах которых создается система.

Качество процесса

Факторы процесса оценивают качество механизмов, применяемых для создания ПО. Они включают:

- **скорость разработки**: способность поставлять продукт в сжатые сроки. Каждый проект должен заботиться об этом, клиенты ждут, конкуренты не дремлют, акционеры размышляют;
- **эффективность стоимости**. Об этом также заботятся почти все проекты. В инженерии программ (в отличие от некоторых других видов инженерии) стоимость готового продукта незначительна. Над всем доминирует стоимость разработки (за исключением, возможно, расходов на маркетинг, которые могут быть существенными, особенно для продуктов, ориентированных на массовый рынок). Стоимость разработки определяют затраты на персонал, оборудование, офис. По этой причине стандартной мерой стоимости является человеко-месяц: средняя стоимость расходов на одного работника в месяц (все включено);
- **эффективность сотрудничества**: эффективность процедур, обеспечивающих наилучшее взаимодействие всех членов команды, работающей над проектом. Серьезные проекты могут включать большое число участников. Требуется уделять особое внимание механизмам координации. Коммуникация участников – довольно деликатная проблема, которая для команд большого размера может перевесить все остальные аспекты разработки. Экстремальная форма этого феномена известна как закон Брукса (по имени проектировщика операционной системы IBM OS/360), который говорит, что «добавление людей в проект, не укладывающийся в сроки, удлиняет время разработки». Хотя можно считать, что этот закон верен только для плохо управляемых проектов, но он ясно характеризует суть проблемы коммуникации;
- **вовлеченность сопричастников:** степень учета в проекте всех релевантных потребностей и точек зрения людей, так или иначе вовлеченных в проект;
- **встроенное оценивание**: включение в процесс механизмов и процедур, измеряющих факторы качества на хорошо определенных этапах. Качество не должно только декларироваться, но его нужно проверять и принуждать к его соблюдению. Хорошо организованный процесс интегрирует эту задачу как один из своих компонентов;
- **предсказуемость**: включение в процесс надежных методов оценивания других факторов качества – в частности, скорости и стоимости разработки. Предсказуемость – одна из наиболее важных характеристик хорошего процесса. Иногда гарантированная дата так же важна, как и ранняя дата. В ИТ-индустрии нет достаточной статистики в этой области, мы не знаем, сколь много проектов не выдерживали сроков и выходили за рамки отведённого бюджета. Ситуация с годами улучшается благодаря применению принципов и методов инженерии программ;
- **измеримость**: пригодность количественных критериев для определения достижимости других факторов качества, как процесса, так и продукта, например, методы измерения уровня ошибок. Эффективное управление нуждается в точных измерениях развития

проекта. Этот критерий тесно связан с двумя предшествующими, так как, чтобы делать предсказания и давать оценки, требуется возможность проведения измерений;

- **воспроизводимость**: независимость разработки, методов управления и предсказания от несущественных атрибутов конкретных проектов. В большинстве случаев индустриальная разработка проекта не ведется изолировано. Крайне важно достигнутые успешные результаты в одном проекте воспроизводить в других проектах (ошибки в проектах также заслуживают тщательного анализа);
- **самосовершенствование**: включение в саму спецификацию процесса механизмов, квалифицирующих и улучшающих этот процесс. Организации, подобно людям, могут обучаться на собственном опыте. Критерий самосовершенствования оценивает, в какой степени процесс, определённый в организации, отвечает этому феномену, за счет включения встроенных механизмов оценки, которые могут служить обратной связью для процесса, адаптируя его в соответствии с уроками обучения.

Модели процессов, такие как CMMI (изучаемые позже в этой главе), ставят эти факторы во главу угла, в частности, последние пять, обучая культуре производства, где оценивание, предсказуемость, измеримость, воспроизводимость и самосовершенствование являются встроенными центральными практиками.

Компромиссы

В то время как разработка должна стремиться к достижению наилучшего качества по всем показателям, предшествующий обзор показывает, что компромиссы неизбежны.

- Компромисс между факторами процесса и продукта: совершенствование продукта может отрицательно воздействовать на фактор «скорость разработки».
- Компромисс между факторами продукта: простота использования не всегда сочетается с безопасностью, так как мы хотим сделать продукт простым для законных пользователей. Пароли плохи с позиций простоты, но хороши для безопасности. Оптимизация в интересах эффективности может конфликтовать с простотой исправлений (так как может приводить к хитро закрученному коду). Эффективность, учитывающая особенности конкретной платформы и контекста, конфликтует с расширяемостью, переносимостью и повторным использованием.

Одна из характеристик хорошо управляемого проекта состоит в том, что компромиссы анализируются и разрешаются явным образом. В противном случае они все равно были бы разрешены как-либо, но далеко не очевидно, что это было бы сделано лучшим образом. Типичный пример – оптимизация, в которой нет очевидной необходимости, но которая может отрицательно влиять на другие факторы.

19.4. Главные виды деятельности в процессе разработки ПО

Инженерия программ включает несколько задач. Вы уже много узнали об одной из их – реализации, и получили некоторое представление о других, таких как проектирование, документирование, задание спецификаций. Теперь мы пройдем по списку главных задач, упорядочим его в первом приближении, начиная от задач, отвечающих интересам клиентов, до тех, что имеют дело с внутренними свойствами ПО.

Анализ осуществимости является задачей изучения проблем клиентов. На этом этапе необходимо принять решение о возможности и желательности построения программной системы (или системы, включающей разработку ПО). Второй аспект, хотя и не следующий из

названия, не менее важен, чем первый, – не всякую систему, которую можно построить, следует строить.

Анализ требований определяет функциональность системы. Элементы, составляющие документ требований, могут быть двух видов.

- *Функциональные* требования, описывающие результаты или действия системы: «Если телефон пользователя переходит из одной области покрытия в другую, то соединение автоматически должно переключаться на точку доступа в новой области».
- *Нефункциональные* требования, специфицирующие ограничения на операции системы. Они включают требования к производительности, например, временные («Для точки доступа, отстоящей менее чем на два километра, время переключения должно занимать не более одной секунды»). Здесь же задаются ограничения на память, безопасность («все сообщения, передаваемые через точку доступа, должны быть зашифрованы»), пропускную способность. Они также покрывают воздействие на системное окружение и людей, таких как служащие: влияние на практику их работы.

Спецификация является точным описанием отдельных элементов системы. Требования ориентированы на клиента. Спецификация преобразует их в форму, непосредственно используемую при разработке системы. Главная разница состоит в законченности и точности: спецификация должна давать недвусмысленный ответ на каждый релевантный вопрос об операциях системы.

Требования и спецификация иногда рассматриваются как единая деятельность. В моделях жизненного цикла, представленных далее, они рассматриваются как независимые. Деятельности, рассматриваемые до сих пор, только указывали на проблему, которая должна быть решена. Для следующих задач мы перейдем в мир программных решений.

Проектирование, называемое также **архитектурой**, состоит в построении общей структуры программной системы. Этот этап ответственен, в частности, за определение отдельных элементов или модулей этой системы и за отношения между этими модулями.

Реализация является задачей фактической разработки программного текста. Она известна также как кодирование, но в термине чувствуется пренебрежительный оттенок – после того, как великие мыслители выполнили анализ и проектирование, рутинную работу по написанию программы могут выполнить прислужники. В этой книге программирование понимается в широком смысле как конструирование программ, не только реализация, но и проектирование и анализ.

Создание документации представляет задачу описания различных аспектов системы, чтобы помочь ее пользователям и другим сопричастникам, в частности, разработчикам. Помимо документов, создаваемых для пользователей, она может включать планы проекта (для менеджеров), документы требований, спецификации, планы проектирования. Слово «документ» заменило более традиционное слово «отчет», представляемый обычно как бумажный документ. Сегодняшняя документация представляется в электронном виде, как Web-страницы, онлайновая справка, регулярно обновляемая. Документация является частью программного текста в виде заголовочных комментариев – специальных документируемых комментариев, которые есть в таких языках, как Eiffel, Java и C#.

Верификация и проверка правильности должны ответить на вопрос, является ли система удовлетворительной. Два аспекта дополняют друг друга.

- **Верификация** является внутренней оценкой согласованности продукта, рассматриваемого сам по себе. Типичный пример: на уровне реализации проверка типов позволяет предотвратить использование переменной, объявленной REAL, как если бы она имела тип INTEGER.

- Проверка правильности дает оценку продукта по отношению свойств, которые должны выполняться: код сравнивается с проектом, проект – со спецификациями, спецификации – с требованиями, документация – со стандартами, применяемые практики – с правилами компании, даты поставки – с намеченными сроками, обнаруженные уровни дефектов – с заявленными целями, набор тестов – со сферой действия метрик.

Популярная версия, характеризующая разницу между верификацией и проверкой правильности, утверждает, что верификация проверяет, «делаются ли вещи правильно», в то время как проверка правильности проверяет, «делаются ли правильные вещи».

Сопровождение, как уже отмечалось, не представляет вид независимой деятельности, а является комбинацией перечисленных выше задач. Единственной отличительной чертой сопровождения является то, что сопровождение начинается с той минуты, когда выпущен первый релиз системы.

19.5. Модели жизненного цикла и разработка в стиле AGILE

В литературе по инженерии программ много внимания уделяется моделям жизненного цикла: спецификации того, как расписать перечисленные выше виды деятельности по фактическим процессам. Подобные упражнения имеют свои пределы, поскольку модели описывают идеализированные процессы, в то время как разработка является человеческой деятельностью со всеми неизбежными элементами непредсказуемости.

Модель водопада

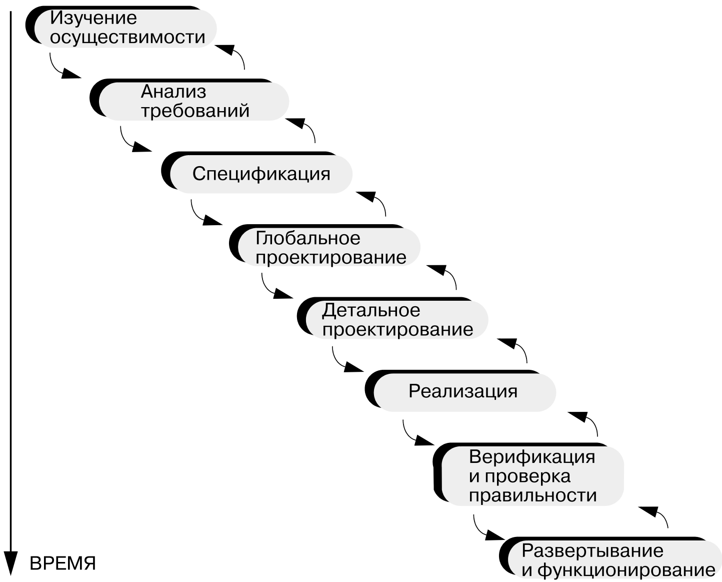


Рис. 19.3. Модель водопада

Начальной точкой всех обсуждений моделей жизненного цикла являлась статья 1970 года, посвященная модели водопада (которая фактически была написана ради критики модели, но служит теперь как ссылка на определение этого понятия). Идея «водопада» проста – задачи выполняются в определенном порядке.

Общей практикой стало графическое представление модели, возможно, из-за отсутствия устойчивого определения. Будем следовать этой практике и здесь. Следующий рисунок задает представление этой модели.

Этот и некоторые другие элементы этого раздела являются адаптированными материалами главы 28 из моей книги «Объектно-ориентированное конструирование программных систем», 2006 г., которая содержит более детальное обсуждение моделей процесса.

Недостаток этой модели – в ее жесткости, так как предполагается, что все виды деятельности синхронизированы. На верхних уровнях модели все может быть гладко и хорошо, но на этапе реализации может оказаться, что надежды не могут быть реализованы в коде, так что приходится возвращаться в начало.

Спиральная модель

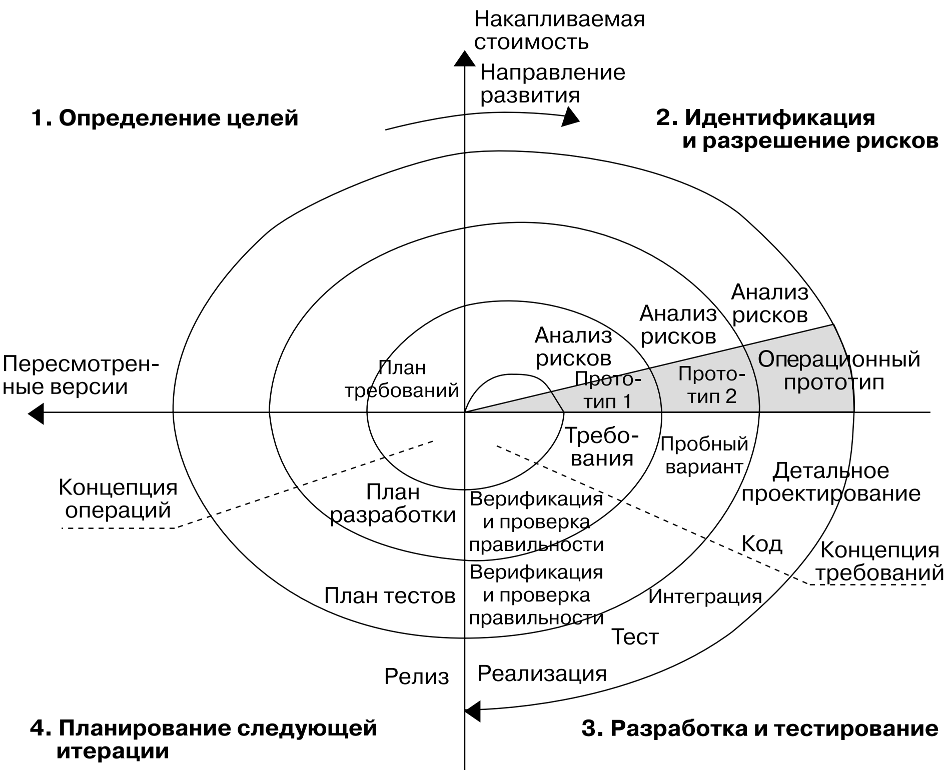


Рис. 19.4. Спиральная модель

В своей книге (Software Engineering Economics Prentice Hall, 1988) Барри Боем предложил модель, которая смягчает жесткость, применяя итеративный подход, основанный на успешных прототипах. Эта модель известна под именем спиральной модели и показана на следующем рисунке.

Каждый прототип в спиральной модели следует последовательности шагов, подобных водопаду, но он предполагает проверку гипотез и пробных проектов, прежде чем создается реально работающая система. Каждая итерация спирали строится с учетом уроков предыдущей итерации.

Спиральная модель более гибкая, чем водопад, и не имеет некоторых принципиальных недостатков. Риск, связанный с этой моделью, состоит в том, что прототип – это не модель, здесь часто ослабляются ограничения (например, производительность), что позже может стать критически важным, перечеркивая ценность уроков прототипа.

Еще один риск, связанный с этой моделью, состоит в том, что когда сокращается бюджет или сроки выпуска релиза становятся определяющими, то в качестве релиза выпускается прототип, изначально не предназначенный для этой цели.

Кластерная модель

Кластерная модель наилучшим образом сочетается с ОО-разработкой в том виде, как она представлена в этой книге. Она модифицирует базисную модель водопада, отменяя синхронность протекающих процессов. Предполагается, что система разделяется на несколько подсистем или кластеров, каждый со своим мини-процессом, как показано на следующем рисунке. В результате к последовательному измерению добавляется аспект параллелизма, так как несколько кластеров могут разрабатываться параллельно. Как следствие, менеджер проекта получает большую свободу в реагировании на сюрпризы процесса разработки. Некоторые задачи будут решены быстрее, чем ожидалось, другие (что чаще происходит) – медленнее.

Для минимизации рисков разработка должна начинаться с наиболее фундаментальных кластеров, обеспечивающих критическую функциональность, и двигаться по направлению к частям, ориентированным на пользователя. Бывает трудно убедить заказчиков в целесообразности такого порядка работы (им бы хотелось иметь работающий прототип как можно скорее), но данный подход чаще приводит к успеху.

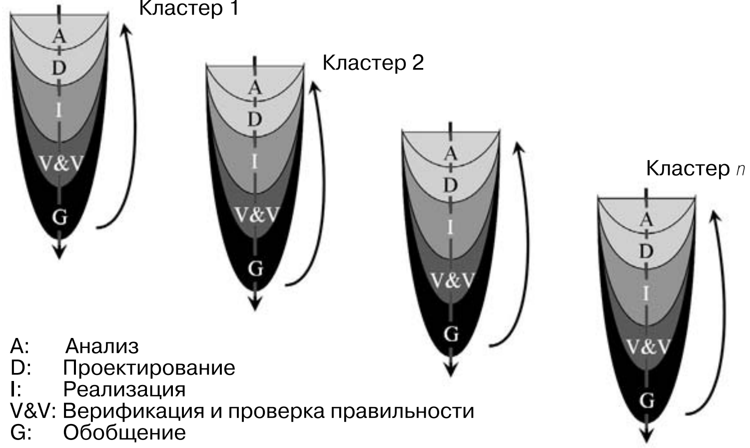


Рис. 19.5. Кластерная модель

Задачи, появляющиеся при разработке каждого кластера, совпадают с задачами водопадной модели за одним исключением – этапа обобщения. Идея в том, что удовлетворительной реализации кластера может быть недостаточно, если нам требуется повторное использование. Целью фазы обобщения является удаление из кластерных классов любого свойства, ограничивающего без необходимости их применимость, такого как ограничение на размеры, несовершенная структура наследования, недостаточность контрактов. В результате кластерные классы могут быть применимыми в будущих разработках и отвечать потребностям текущего проекта.

Разработка каждого кластера выполняется непрерывно, а не в виде последовательности независимых шагов, как в модели водопада. Эта идея бесшовной разработки, в частности, важна в подходе Eiffel, где анализ, проектирование и реализация – все используют единую нотацию и построены на едином базисе, начиная с отложенных классов высокого уровня, которые описывают проблему. Фазы проектирования и реализации состоят из уточнений и обогащений этих классов. Это облегчает возврат (символизируемый стрелкой возврата на рисунке) к предыдущим фазам для улучшения или корректировки несовершенной предыдущей версии.

Agile – гибкая методология разработки

В ответ на жесткость процессов, предписываемых моделью разработки, agile-методы, в частности, подход, называемый «экстремальным программированием», меньше внимания уделяет планам и процессам, фокусируясь вместо этого на элементах, таких как:

- работающий код как принципиальная мера прогресса разработки;
- сотрудничество между разработчиками и заказчиками, которые имеют своих представителей в команде разработчиков;
- частые коммуникации;
- тесты (а не спецификации) руководят разработкой;
- небольшие добавления в проект обеспечивают постоянную обратную связь и непрерывную интеграцию: непосредственная компиляция и тестирование изменений; незамедлительная их интеграция в основной проект, не ожидая недель и месяцев, уменьшая риск того, что две части проекта разойдутся и их слияние будет затруднительным;
- специфические практики, такие как «парное программирование» (разработчики разбиваются на пары и создают код вместе, что позволяет раньше обнаруживать ошибки).

Появление в девяностые годы этой гибкой методологии вызвало ожесточенные споры. Это воспринималось как социологический феномен – революция программистов против засилья менеджеров. Теперь все понемногу успокоилось, и многие практики, такие как непрерывная интеграция, широко приняты. Другие, такие как «тесты предпочтительнее спецификаций», остаются под вопросом. Но становится ясно, что процесс разработки ПО может быть как структурированным, так и гибким (agile).

19.6. Анализ требований

Без подходящих программистских приемов: алгоритмов, структур данных, контрактов, анализа производительности, модульной структуры, поддержки компилятора, поддержки инструментария и других изученных приемов – без всего этого проекты потерпят неудачу. Необходима технология, но ее недостаточно. Успешные системы строятся в интересах клиентов и должны соответствовать их потребностям. Анализ требований позволяет достигнуть хорошего соответствия между тем, что хотят пользователи, и тем, что делает система.

В этом одна из центральных задач разработки. Она трудна, но может доставлять радость. Самое время развеять взгляд на программистов как на погруженных в себя зануд. На самом деле практика показывает, что они много времени проводят в обсуждениях с пользователями и другими нетехническими участниками.

Следующий обзор описывает некоторые из вызовов, с которыми приходится сталкиваться, и несколько принципов, которые следует учитывать при разработке эффективных требований.

Продукты на этапе требований

Процесс создания требований должен выдавать два конкретных результата:

- документ требований, описывающий характеристики системы, которую предстоит построить;
- В&П-план (план верификации и проверки правильности), часто называемый «тест-план», описывающий, как построенная система будет оцениваться.

Второй продукт часто игнорируется, но он так же важен, как и первый. Пришло время серьезных проектов, когда перед построением ПО создается хороший В&П-план, а еще лучше – хороший страховочный план. Создание тестов на этом этапе направлено на оценку соответствия системы заявленным намерениям. Чем позднее создаются тесты, тем вероятнее, что они будут руководствоваться решениями, принятыми на этапе проектирования и реализации, и в меньшей степени – потребностями клиентов.

Стандарт IEEE

Существует полезный ресурс для подготовки требований: стандарт IEEE Computer Society (совместно с ACM – одной из наиболее активных профессиональных ассоциаций ИТ. Мы уже упоминали один из ее стандартов для арифметики с плавающей точкой). Стандарт «Recommended Practice for Software Requirements Specifications» («Рекомендуемая практика специфицирования требований ПО») определяет некоторые лучшие способы задания требования, включая универсальную структуру документа требований.

Это краткий и простой стандарт. С ним стоит познакомиться, а если необходимо написать требования – то следовать рекомендуемой структуре, широко используемой в индустрии. Эта структура состоит из трех частей: введения, общего описания и специфических требований. Часть 2, помимо описания, включает перспективу продукта, функции продукта, характеристики пользователя, ограничения, предположения, зависимости, выделенные требования. Часть 3 идет глубже, уделяет внимание таким деталям системы, как внешние интерфейсы, требования к производительности, требования к базам данных.

Все это – не более чем контрольный список свойств системы, указываемый в требованиях. Поскольку многие из этих свойств могут влиять на успех разработки, следующий стандарт помогает авансом избежать дорогостоящих ошибок.

Область требований

Программная система почти всегда является частью большой системы. Встроенное ПО, скажем, в цифровую камеру или мобильный телефон, является частью системы, включающей аппаратуру. Бизнес-ПО – это часть системы, включающей процессы компании. Одно из первых решений, которое приходится принимать при подготовке требований, – должны ли они относиться только к ПО или ко всей системе. Первый ответ не означает игнорирование системы, он предполагает, что будут заданы четкие интерфейсы между ПО и его окружением.

Еще одно важное разграничение, отмеченное ранее, – разграничение функциональных и не функциональных аспектов.

- Функциональные требования специфицируют ответы системы. «Если ввод в поле номера социальной службы является правильным номером, то система должна отобразить имя и фамилию соответствующей персоны» – это функциональное требование.
- Нефункциональное требование специфицирует все другие свойства системы, такие как производительность, доступность, простоту использования. «Отображение имени и фамилии не должно занимать более чем 0,2 секунды в 99, 5% случаев» – это пример нефункционального требования.

Обратите внимание на терминологию: «требование» – это элемент специфицируемого поведения, функционально или нефункционального, как в приведенных примерах. Требования – это коллекция таких индивидуальных элементов.

Получение требований

Процесс получения (или извлечения) требований системы может широко варьироваться. Он может быть совершенно неформальным, когда несколько человек, понимая основы системы, быстро трансформируют их в фактическую разработку. Подход «agile», упоминаемый выше, предпочитает вместо тяжелого предварительного процесса задания требований постоянное взаимодействие с заказчиками. Однако многие большие индустриальные компании прилагают значительные усилия по созданию вначале требований, не исключая при этом последующего их пересмотра при появлении новых идей в процессе конструирования системы.

Этот последний комментарий проясняет общее свойство сбора требований. Вы, возможно, полагаете, что в идеале процесс полностью создается на основе «потребностей пользователей»: некоторая команда внимательно опрашивает клиентов, выясняя, что они хотят, записывает их ответы, сортирует, создает документ требований и вручает его команде разработчиков, которая и реализует желания клиентов. Так почти никогда не происходит. Так и не должно происходить.

- Сопричастники часто имеют конфликтующие интересы – кто-то должен разрешать конфликты.
- Требования пользователей часто представляют смесь простых, осуществимых, трудных (или невозможных) свойств. У пользователей часто нет понимания, что является простым для реализации, а что трудным.
- Только команда разработчиков может оценить техническую стоимость каждой запрашиваемой функциональности; основной критерий принимаемого решения – включать ли данное требование.
- Пользователи имеют тенденцию думать в терминах существующих систем (или, в другом экстремальном случае, мечтают о системах, которые невозможно построить). Чаще всего именно разработчики могут предложить нового типа функциональность, которую пользователи не могли даже вообразить. Это общее свойство технологических инноваций: немногие прорывные продукты были спроектированы на основе пожеланий, собранных у пользователей. Технологи обычно выслушивают пользователей, а потом возвращаются к ним с собственным предложением: а что, если я дам вам следующее устройство?
- Многие внешние факторы влияют на окончательный выбор функциональности, такие как бюджет, существующая система, интерфейс с другими системами.

Наблюдения показывают, что помимо простейшего случая сбора пожеланий клиентов и их последующей реализации, обычный процесс – итеративный: пользователи высказывают

пожелания, разработчики предлагают легко реализуемые свойства и после нескольких итераций останавливаются где-то посередине. Такие обсуждения могут носить креативный характер, в результате могут появиться требования, которые ранее ни одна из групп и не предполагала.

Процесс принятия требований должен поддерживать эту модель. Практическое правило состоит в том, что разработка не должна начинаться до одобрения документа требований. Он должен быть подписан заказчиком с правом ответственной подписи и руководителем группы разработки. Многих ошибок в проектах можно было бы избежать, придерживаясь этого простого правила.

> Иногда компания создает документ требований, не имея команды разработчиков, внешней или внутренней, так что описанный процесс становится неприменимым. Идея в таких случаях понятна – вначале определить требования, а потом выбрать команду, наилучшим образом реализующую требования. Это рискованная практика. Она может приводить к нереализуемым требованиям. Риск особенно возрастает, если на этом этапе привлекать внешних консультантов для проведения анализа требований. Мне неоднократно доводилось видеть для индустриальных проектов, что такой процесс приводит к завышенным требованиям. Слишком велико желание сделать приятное заказчикам, давая обещание, выполнять которое должен кто-то другой. В таких ситуациях в лучшем случае требования будут неизбежно пересмотрены, в худшем – реализация приведет к задержкам или неудаче.

Приемы для сбора требований включают:

- интервью. Представители разных категорий сопричастников опрашиваются с целью выяснения, что они ждут от новой системы или от изменения существующей. Интервью должны быть тщательно подготовлены, включать вопросы по отдельным разделам и иметь открытую часть для пожеланий в свободной форме. Общей практикой является видеозапись интервью, чтобы к нему можно было бы возвращаться при необходимости;
- семинары. Сбор в одном месте опрашиваемых людей может быть лучше, чем индивидуальные интервью. В этом случае часто на месте удается устранить разногласия, возникающие у разных групп сопричастников;
- предыдущие системы. Редко, когда система начинается на пустом месте. Обычно уже существует система, отвечающая тем же потребностям. Изучение существующих систем, их достоинств и недостатков, является важной частью сбора требований. Одним из технических требований в таких ситуация может быть требование, чтобы новая система, по меньшей мере, работала не хуже старой и давала те же результаты в сравнимых случаях;
- конкурирующие системы. Необходимо знать, что предлагают конкуренты. Даже если речь идет о внутренней разработке, полезно знать, как справляются с этим конкуренты, имеющие те же потребности.

Глоссарий

Одним из продуктов, входящим в состав требований, должен быть глоссарий (в стандарте IEEE это раздел 1.3 «Определения, акронимы, аббревиатуры»).

Каждая проблемная область имеет свой жаргон. Эксперты в этой области используют его в интервью и семинарах, предполагая, что интервьюер понимает его и что остальные эксперты понимают его так же, как и они. Оба предположения могут оказаться неверными. Первой

задачей документа требования является перечисление употребляемых терминов и определение их точного технического смысла. Соберите все такие определения в глоссарий, покажите его экспертам, чтобы убедиться, что они согласны с вами и друг с другом. Глоссарий является одним из принципиальных ресурсов для процесса разработки требований. Но он используется не только для этих целей, поскольку многие концепции, перечисленные в глоссарии, будут необходим прямым двойникам (классам, компонентам) в программах.

Свойства проблемной области и машинные (системные) свойства

При написании требований всегда следует четко разграничивать машинные и проблемные свойства (о чем убедительно сказано в книге Майкла Джексона, см. ссылку в конце главы).

Рис. 19.6. Майкл Джексон (2004)

Любая система функционирует в некоторой проблемной области со своими законами: в электронике действуют физические ограничения на скорость сигнала, в банковской сфере – своя система правил. Программная система, появляющаяся в результате разработки, может рассматриваться, как уже говорилось, как своего рода машина со своими законами. Джексон указывает на необходимость разграничений двух категорий требований.

- Никакая передача не должна быть принятой, если в результате баланс счета становится ниже установленного предела. Это проблемное свойство, оно следует в данном случае из правил, принятых в бизнесе.
- Любая передача, в результате которой баланс счета становится ниже установленного предела, должна приводить к посылке сообщения менеджеру счетов. Это машинное свойство, которое описывает частное решение, принятое в системе (в фактическом документе требований оно должно быть сформулировано более точно). Данное машинное требование непосредственно следует из приведенного проблемного требования. Не все машинные требования являются следствиями проблемных. Некоторые отражают чисто системные решения.

В кратком тексте, описывающем Парижское метро, утверждение «У метро есть важное свойство – всегда существует маршрут для любой пары станций метро (математики бы сказали, что метро задается *связным* графом)» – является проблемным свойством. Любая программная система, связанная с метро, должна гарантировать выполнение этого свойства. Правило, нумерующее станции с юга на север (явно введенное «для упрощения жизни»), является машинным правилом, которое описывает частное соглашение, выбранное в конкретной модели метро.

Помимо необходимости понимания проблемной области возникает еще одна новая задача, отличающаяся от инженерии требований, – инженерия проблемной области, направ-

ленная на моделирование общих свойств проблемной области. Инженерия проблемной области не связана с конкретным проектом, но помогает процессу задания требований для всех проектов данной проблемной области. Например, компания, которая регулярно разрабатывает проекты, связанные с управлением движения поездов, может инвестировать средства в моделирование общих свойств железнодорожных систем.

Требования являются комбинацией машинных и проблемных ограничений. Слишком часто документы требований не разграничивают эти два вида требований. В результате читатели документа, в частности, разработчики, не понимают четко, что является следствием внешних обстоятельств (скорость света изменить нельзя), а что может быть пересмотрено при эволюции системы. По этой причине важно в документе требований специфицировать их природу – машинную или проблемную – для каждого частного требования.

Пятнадцать свойств хороших требований

Давайте дополним наш обзор рассмотрением свойств – пятнадцати свойств, – которым должны удовлетворять хорошие требования. Рассмотрим требования к требованиям. Должен заметить, что я никогда не видел документ, удовлетворяющий всем этим свойствам, но они обеспечивают ясное понимание принципов, которым должен следовать каждый разработчик требований. Некоторые, но не все, отвечают стандарту IEEE, ниже они отмечены звездочкой.

Требования должны быть **обоснованными**. Каждое индивидуальное требование должно иметь источником идентифицируемую и явно установленную потребность сопричастника.

Требования должны быть **корректными** *. Любая система, удовлетворяющая требованиям, должна отвечать потребностям сопричастников. Формально это невозможно гарантировать. Неформально нужно быть уверенными, что все сопричастники знакомы с требованиями и согласны с относящимися к ним требованиями.

Требования должны быть **полными.** Они должны покрывать все потребности сопричастников. В принципе, полнота не проверяема, так как возникает естественный вопрос – полнота по отношению к чему? Любой ответ будет ссылаться на некий более высокий уровень, новый документ, куда и переносится решение проблемы. На практике существуют полезные эвристики, основанные на концепциях, введенных ранее в этой книге.

Почувствуй методологию

Достаточная полнота

Документ требований должен определять эффект каждой команды системы для каждого запроса системы.

Подобно классу, каждая система обеспечивает команды и запросы. Можно выполнять некоторые действия и можно запрашивать информацию. Документ требований должен описывать как команды, так и запросы. Информации должно быть достаточно, чтобы читатель требований мог определить, как выполнение любой команды будет воздействовать на любой из доступных запросов.

«Достаточная полнота» – технический термин, введенный в 1978 году в статье Гуттага и Хорнинга для характеристики свойств абстрактных типов данных, теоретической основы ОО-программирования.

Требования должны быть **согласованными**. Они не должны включать противоречий. Этого на удивление трудно достигнуть. Трудность частично связана с размером многих индуст-

риальных документов требований, которые могут составлять сотни и тысячи страниц для сложных систем. Несогласованности прокрадываются в документ. На странице 235 утверждается, что шлагбаум должен опускаться *до* звукового сигнала, свидетельствующего о приближении поезда, а на странице 1232 утверждается обратное. Программисту нужно выбрать что-то одно. На этапе принятия требований несогласованности должны быть обнаружены и устранены.

> Заметьте разницу между согласованностью и корректностью: согласованность – внутреннее свойство документа требований, корректность указывает на удовлетворение некоторым внешним ограничениям. Здесь такое же разделение, как между верификацией и проверкой правильности.

Требования должны быть **недвусмысленными**. Задача трудновыполнимая, поскольку документы требований пишутся на естественном языке с присущей ему неоднозначностью. Рассмотрим пример:

Фоновый менеджер задач должен обеспечивать сообщения о статусе с регулярными интервалами, не превышающими 60 секунд.

Пример взят из «SoftwareRequirements», см. ссылку в пункте 19.9 «Дальнейшее чтение».

Разработчики системы по-разному могут истолковать это требование, некоторые интерпретации могут не устраивать пользователей. Эксперт по требованиям, процитировавший этот фрагмент, предложил в качестве замены некоторые варианты (сделав некоторые предположения о намерениях, которые можно было бы проверить у пользователей).

1. Фоновый менеджер задач (ФМЗ) должен отображать сообщения о статусе в спроектированной области интерфейса пользователя.
2. Сообщения должны обновляться каждые 60 (плюс или минус 10) секунд после начала обработки фоновой задачи и должны постоянно оставаться видимыми.
3. Если обработка фоновой задачи выполняется нормально, то ФМЗ должен отображать процент выполненной части фоновой задачи.
4. ФМЗ должен отображать сообщение «Выполнено» по завершении фоновой задачи.
5. ФМЗ должен отображать сообщение об ошибке, если фоновая задача снята с выполнения.

Это значительно более точный и типичный стиль для индустриальных требований. Пример является хорошей иллюстрацией тех трудностей, которые возникают при задании требований, он позволяет понять, почему тщательно написанный документ требований может занимать тысячи страниц.

Естественные языки не способствуют точности описаний. По этой причине во многих работах предпринимаются значительные усилия по использованию математических приемов для задания требований, называемых в этом случае *формальными спецификациями*.

Статья о «Формализме в спецификациях», см. ссылку в конце главы в разделе «Дальнейшее чтение», обсуждает преимущества и недостатки такого подхода.

Требования должны быть **осуществимыми**. Вполне возможно «витание в облаках» при задании требований, особенно, как отмечалось, если требования пишут не те, кто будет их реализовать. Серьезный процесс включает ограничение амбиций и понимание возможностей.

Требования должны быть **абстрактными**. Типичный просчет при подготовке требований состоит в попытках задания решений, которые следует принимать на последующих этапах проектирования и реализации. Такая сверхспецификация сужает возможности и не соответствует миссии требований, которые должны говорить, *что* нужно *делать*, но не определять, *как делать*.

Требования должны быть **прослеживамыми***. Другими словами, должна быть возможность проследить в коде и в других программных продуктах все последствия каждого индивидуального требования. Это позволяет не только проверить, чтобы предлагаемая реализация удовлетворяла всем требованиям, но и при изменении требования проследить за всеми программными элементами, затронутыми этим изменением.

Примером механизма прослеживания в EiffelStudio является средство, названное EIS (Eiffel Information System), поддерживающее определение связей между индивидуальными элементами документа требований и классами и компонентами программной системы. В принципе, каждый программный элемент должен быть прямым или косвенным следствием того или иного требования, а каждое требование должно иметь двойника в программной системе. Механизм EIS позволяет добавлять связи в документ PDF или MicrosoftWord, так что щелчок по связи приводит к открытию EiffelStudio на соответствующем участке кода – спроектированном классе или компоненте. Возможно и обратное действие: добавить связь в код, переход по которой приведет к соответствующему требованию. Механизм EIS является прямой реализацией принципа прослеживания, предназначенной, в частности, для облегчения процесса внесения изменений в требования.

Требования должны быть **верифицируемыми***. Бесполезно задавать требование, если нет способа проверки его выполнения в программной системе. Экстремальным – но, к несчастью, распространённым – примером неверифицируемого требования является требование в форме «система должна работать в реальном времени, выполняя команды и запросы». Но что такое «реальное время», может оставаться загадкой. Для банковской системы ответ на запрос в течение 2-х секунд – это реальное время, для сетевых устройств реальным временем может быть 100 микросекунд. Документ должен указывать точные значения, задавая среднее значение и допустимые отклонения.

Требования должны быть **ограничительными**. Важно установить не только то, то система должна делать, но и то, что лежит за пределами ее компетенции.

Требования должны **задавать интерфейс**. Следует точно установить связи системы с внешним миром – людьми, аппаратурой, другими программными системами.

Требования должны быть **приоритетными***. Иногда обстоятельства заставляют отказаться в проекте от полной реализации всего, что было запланировано. Типичной причиной отказа является урезание бюджета.

Причиной могут стать неожиданно возникшие трудности, приводящие к задержке проекта, и, как следствие, ограничение функциональности, чтобы проект мог выйти в запланированные сроки. Иногда приходится форсировать выпуск, чтобы опередить конкурирующий продукт. Выбор того, чем следует пожертвовать, не должен приниматься спонтанно. Требования должны устанавливать важность каждой функциональности, это позволяет делать выбор на основе предварительно согласованных приоритетов.

Требования должны быть **понятными**. Стремление к точности и детализации может в результате дать обратный эффект, приводя к громоздким документам. Если требования не будут просты для понимания, они не сыграют свою роль.

Требования должны быть **модифицируемыми***. Меняются обстоятельства, меняется сознание людей, компании могут сливаться. Подобно любым другим программным продуктам, требования должны проектироваться с учетом возможных изменений.

Ну и, наконец, требования должны быть **одобрены и подписаны**. Так много места для непонимания и конфликтов, что не следует начинать разработку проекта без ясного формального понимания, включающего, по крайней мере, подписи людей, ответственных за проект с двух сторон – со стороны заказчика и со стороны команды разработчиков.

Надеюсь, я не напугал вас этим длинным списком критериев хороших требований. Документ хороших, хотя и не совершенных требований написать вполне возможно, он будет отображать потребности сопричастников проекта и будет служить основой для разработки и В&П-реализации. Это важная часть разработки ПО и великолепная возможность комбинировать технологию с бизнесом, учитывая при этом психологию людей.

19.7. В&П – Верификация и проверка правильности

Первое правило В&П говорит, что было бы прекрасно, если бы этого не нужно было делать. Цель всех правил проектирования и методологии программирования, изучаемых в этой книге (а я верю, что вы будете применять каждое из них при каждом подходящем случае), состоит в том, чтобы создавать программный продукт, который будет работать с первого раза и каждый раз. Но все же нужно убедить в этом остальной мир, не исключая возможности, что ошибки могут встретиться и в вашей работе. Кроме того, иногда ведь приходится модифицировать продукт, написанный другим, менее просвещенным программистом. На практике В&П является главной частью усилий при разработке проекта, часто занимающей больше времени, чем само конструирование ПО.

Ограничим себя обзором некоторых базисных идей. Обсуждение главным образом коснется программ, хотя, как отмечалось, В&П применима и к непрограммным артефактам, таким как документация. Термин «страхование качества ПО» будет использоваться как синоним В&П.

Это некоторое насилие над языком, так как страхование качества включает методы построения качественного ПО и методы оценивания качества построенного ПО.

Разнообразие страхования качества

Многие люди под В&П понимают только тестирование и отладку. Фактически, область применения методов В&П значительно шире. Тестирование – главный вид *динамических* методов – заключается в выполнении системы (поэтому и динамический метод) на выбранных входах. Цель тестирования – обнаружение дефектов.

Статические методы анализируют текст программы без ее выполнения. Они включают осмотр кода, его статический анализ, доказательство корректности, проверку на соответствие модели.

Наше рассмотрение начнем с тестирования, а затем перейдём к статическим приемам. Следующие термины, полезные при обсуждении, идут от еще одного IEEE-стандарта по терминологии инженерии программ.

IEEEStd 610.12-1990, tinyurl.com/3w57pk (текст 1990 года, но во многом все еще полезен).

- Выполнение программы, не функционирующее, как ожидалось (дает неверные результаты или приводит к аварийному завершению), называется **отказом** (**failure**).
- Отказ (за исключением редких случаев отказа аппаратуры) – следствие **дефекта** (**fault**) ПО, свидетельствующее, что ПО работает не так, как должно. Заметьте, что дефект не обязательно является ошибкой реализации, он может возникать из-за ошибок на любом уровне, таких как спецификация или проектирование.
- Дефект является следствием **ошибки** (**mistake**), сделанной разработчиком ПО.

Термин «баг», или «жучок», не является частью официальной терминологии, хотя часто используется для обозначения дефектов или ошибок чаще всего при *отладке* – задаче исправления ошибок, удаления дефектов и устранения отказов.

Тестирование

Начнем с тестирования – наиболее применяемой программистами техники. Первое наблюдение свидетельствует о скромной роли тестирования, поскольку оно не дает гарантий качества. Причина была указана Эдсгером Дейкстрой, это его высказывание является одним из наиболее цитируемых в истории информатики: «Тестирование может показать наличие ошибок, но не их отсутствие».

Ошибочный тест выявляет дефект, успешный тест мало о чем говорит, так как в любой реальной программе число тестов невероятно велико. Даже программа, которая умножает два числа из 64-х битов, имеет 2^{128} вариантов.

Комментарий Дейкстры справедлив, но он не свидетельствует о грехах тестирования. «Показать наличие ошибок» – дело, чрезвычайно полезное для практики, позволяющее нам найти ошибки до того, как их найдут пользователи. Тестирование – это техника обнаружения ошибок.

За последние годы технология тестирования существенно прогрессировала. Эволюция шла в направлении большей автоматизации. Для всех известных языков программирования появились специальные инструментальные средства – каркасы, позволяющие записывать тесты и автоматически запускать систему тестов. Эти средства известны под общим именем «XUnit», следуя названию JUnit – каркасу Java. Их широкий успех объясняется тем, что альтернативный способ – ручное управление и выполнение тестов – стал нереалистичным для современных программ, требующих запуска большого числа тестов. Мощь компьютеров сделала возможным проверку системы на многих тестах, но для этого требуется автоматическая поддержка.

Автоматизация, в частности, необходима для задачи, известной как регрессионное тестирование. Имеет место факт, удивительный для новичков, но характерный для разработки ПО,– исправленные дефекты могут вновь появляться в последующих релизах (ПО частично регрессирует к прежнему состоянию, отсюда и название – регрессионное тестирование). Причины регрессии следующие:

- неудачно проведенная коррекция, устраняющая симптом, но не причину (настоящую ошибку);
- может существовать некоторый образец ошибки, который становится причиной появления разных отказов. Корректировка некоторых из них не устраняет причину полностью.

Регрессионное тестирование пытается захватить все такие случаи, выполняя все тесты, на которых возникала ошибка. Каждый серьезный проект выполняет регрессионное тестирование при каждом новом выпуске системы. Это отражается в следующем принципе.

Почувствуй методологию

Каждый ошибочный тест должен стать частью набора регрессионного тестирования и оставаться в наборе на протяжении всей жизни проекта.

Современные исследования автоматизации тестирования идут значительно дальше. Примером новых возможностей является каркас – EiffelTestFramework, который, начиная с версии 6.3, стал интегральной частью EiffelStudio. Здесь в добавление к стандартному механизму «XUnit» появились два продвинутых свойства.

- Синтез теста из отказа: каждое выполнение, приводящее к отказу, в соответствии с принципом ошибочного теста автоматически создает тест. Новинкой здесь является автоматизация. Многие из наиболее важных потенциальных тестов приходят из интер-

активных выполнений, которые приводят к отказам в процессе разработки, но при обычном подходе теряются после коррекции и не появляются в наборе регрессионного тестирования. Здесь же процесс преобразования отказа в воспроизводимый тест осуществляется автоматически.
- Генерация тестов по спецификациям: можно сделать запрос к тестирующему каркасу на тестирование класса, не задавая при этом никаких входных данных. Инструментарий автоматически будет проверять все методы класса, используя значения и объекты, создаваемые автоматически. Процесс может идти в фоновом режиме, в то время как вы можете продолжать работать над проектом. Вас потревожат только в случае обнаружения ошибки. Помните, что тестирование служит не для выяснения качества, а для обнаружения ошибок. Ошибки в данном случае будут возникать при нарушении постусловий и инвариантов.

Тестирование – активно развивающаяся область исследований, так что можно ожидать появления новых инструментариев и новых возможностей в будущих средах разработки.

Возвращаясь назад, к сегодняшней технологии тестирования, стоит рассмотреть несколько новых понятий (за деталями следует обратиться к учебникам по инженерии программ и литературе по тестированию).

Тестирование встречается на разных уровнях гранулярности.

Юнит-тестирование (модульное тестирование) предназначено для тестирования отдельных модулей – типично классов или кластеров в ОО-разработке. Обычно оно выполняется индивидуальными разработчиками соответствующих модулей.

Интеграционное тестирование оценивает, как выполняется группа модулей или подсистем при их соединении. Обычно это задача группы разработчиков, возможно, выполняемая специализированным подмножеством – командой тестеров или командой страхования качества.

Системное тестирование тестирует систему в целом. Часто этот шаг по-прежнему выполняется командой тестеров. *Приемочное тестирование* предполагает тестирование с позиций заказчика, и ответственность за него лежит на организации заказчика, оно выполняется объединенной группой представителей разработчиков и заказчика.

Для юнит-тестирования принято различать два подхода – черный и белый ящики. При тестировании белого ящика доступен текст программы, позволяющий руководить процессом тестирования, в то время как тестирование черного ящика целиком основано на спецификациях модуля. Тестирование черного ящика – единственно возможный способ, когда модуль приходит от внешнего поставщика и необходимо оценить его применимость в вашей разработке. Оно может представлять интерес и в том случае, когда текст доступен. Примером такого подхода является упомянутый механизм тестирования EiffelTestFramework, когда автоматически строятся тесты по спецификациям класса с использованием контрактов.

В заключение отметим концепцию *покрытия тестов*, применимую главным образом к белым ящикам. Покрытие – это мера качества набора тестов, позволяющая оценить объем протестированной функциональности. Меры покрытия могут включать:
- покрытие операторов: какой процент операторов программы выполнялось при запуске набора тестов?
- покрытие ветвей: какой процент ветвей программы (элементарных путей программы, например, каждый условный оператор задает две ветви) был пройден при запуске набора тестов?

Существует много других критериев покрытия, хотя в конечном итоге считать нужно было бы, каков процент обнаруженных ошибок дает данный набор тестов. Этот критерий может коррелировать или не коррелировать с элементарными мерами покрытия. Обобщение

черного ящика приводит к понятию покрытия спецификации, дающего оценку, как много случаев, допускаемых спецификацией, были испытаны.

Статические методы

Закончим обзор В&П рассмотрением статических методов.

Обзор проекта и кода, называемый также инспекцией, является процессом, выполняемым вручную для обнаружения дефектов и других неисправностей. Целью инспекции типично бывает программный элемент, за разработку которого отвечает один исполнитель. Таким элементом может быть класс, но может быть и глава из руководства пользователя. Текст пускается по кругу, а затем обсуждается на встрече в интересах обнаружения возможных проблем. Встреча не имеет других целей – не оценивается сам разработчик и не предполагается исправление обнаруженных проблем (эту задачу будет решать разработчик самостоятельно после встречи).

Это описание классической идеи обзора кода. С наступлением эры Интернета и все большим распространением географически распределенных команд разработчиков этот процесс может использовать возможности удаленного доступа и видеоконференций. Один из уроков такого опыта состоит в том, что обзор становится более эффективным, если частично сопровождается написанием замечаний. Процесс начинается еще до встречи, когда участники аннотируют общий документ (технология разделения Web-документов теперь широко доступна). По большинству замечаний на этом этапе разработчик и его критики приходят к согласию. На встречу (видеоконференцию) выносятся вопросы, требующие дополнительного обсуждения.

Моя статья «Design and Code Reviews in the Age of the Internet» (Communications of the ACM, vol. 51, no. 9, September 2008) описывает этот процесс в деталях.

Нельзя ожидать, что инспекция кода является эффективным инструментом для систематического обнаружения дефектов. Обзор кода является процессом, требующим больших затрат времени разработчиков – самого дорогого ресурса. Хорошей идей является применение этого подхода для критически важных модулей. Главная же цель выполнения обзоров состоит в оценке общих практик проектирования и кодирования, применяемых в команде, особенно практик, способных повредить качеству системы. Крайне важно, когда обзор выявил недостатки, выявить их причины и понять, какие же методы могут быть использованы, чтобы избежать подобных ошибок в будущем.

Более эффективный процесс статического анализа требует автоматизированного инструментария. Компилятор статически типизированного языка включает статический анализатор, являющийся частью компилятора и обеспечивающий безопасность системы типов. Помимо прямой реализации правил языка программирования статический анализатор ищет образцы кода, которые могут приводить к ошибкам, даже если они явно не нарушают правила языка. Примеры включают:

- переменные, которые на некоторых путях могут стать доступными ещё до того, как они получили значения (в языке, не предусматривающем автоматической инициализации);
- неиспользуемые переменные (не ошибка, но аномалия);
- void-вызовы (если язык не гарантирует void-безопасность).

Специальной формой статического анализа является доказательство корректности программ – наиболее амбициозный и наиболее трудный подход. Термин «доказательство» используется в математическом смысле и, следовательно, предполагает некий формализм при задании спецификаций. Контракты Eiffel дают представление о том, как может выглядеть такой формализм: каждый программный элемент характеризуется предусловием и постус-

ловием (для методов) или инвариантом (для класса). Это абстрактные спецификации функциональности. Для полного доказательства спецификации должны быть детализированы, но общая идея остается применимой. «Доказательство» класса тогда означает установление математическими методами, что реализация удовлетворяет спецификации: каждый метод, запущенный в состоянии, удовлетворяющем инварианту и предусловию, будет завершать свое выполнение в состоянии, удовлетворяющем постусловию и инварианту.

Эта форма спецификаций согласована с наблюдением, приведенным ранее в этой главе, что корректность программы может быть определена только по отношению к заданной спецификации. Здесь спецификации принимают форму контрактов, а корректность означает, что реализация согласована с контрактами.

Поскольку такие доказательства требуют включения многих деталей, а также и потому, что доказательствам, сделанным человеком, нельзя полностью доверять (ошибки в доказательствах возможны не в меньшей степени, чем ошибки в программе), процесс должен основываться на автоматически работающем инструментарии, выполняющем доказательство корректности программ. Многие такие программы представляют надстройку над программами, доказывающими правильность теорем, способных выполнять математические выводы. Работы над автоматизацией методов доказательства ведутся десятилетиями; в последние годы они получили новые импульсы благодаря продвижениям в технологии доказательств и лучшего понимания проблем. Это активно развиваемая область исследований.

Наиболее впечатляющий прогресс достигнут в направлении, где обычно не пытаются дать полное доказательство корректности, но фокусируются вместо этого на идентификации специфических отказов, – то, что обычно делает тестирование. Проверка модели использует преимущество мощи компьютера для исследования пространства состояний программы или, более реалистично, ее упрощенной версии – модели. Если это позволяет уменьшить пространство состояний до приемлемых размеров, то тогда можно определить, будет ли нарушать любое из состояний интересующее нас свойство – часто корректность или безопасность. Этот подход интегрирует некоторые идеи тестирования (исследование многих случаев, фокусируясь на непокрытых дефектах вместо установления полной корректности). Но в целом это статический метод, являющийся формой доказательства. Абстрактная интерпретация определяет абстрактную версию программы, к которой применимы продвинутые методы статического анализа. Одним из успешных проектов было доказательство большой критически важной программы, встроенной в «Аэробус» A330/340 и A380, в результате чего не было отказов системы в период ее эксплуатации.

Что это дает, можете вы спросить, для повседневной практики программирования? Это во многом зависит от места вашей работы. Долгое время формальные методы – доказательства и связанные приемы – рассматривались как интеллектуально привлекательные идеи, не применимые в индустриальных разработках (назвать подход «академическим» равносильно «поцелую смерти»). Эта точка зрения сегодня уже не доминирует. При постоянном улучшении как теории, так и инструментария, и с возрастающей опасностью рисков неверно функционирующих программных систем растет число индустриальных разработок, использующих формальные методы и инструменты. Некоторые из уроков обнадеживают, некоторые разочаровывают.

- Положительно то, что формальные инструменты работают. Можно разработать реальную систему, поставляемую с полной гарантией корректности. Кстати, заметьте, что такие доказательства не говорят о совершенстве ПО, утверждается только, что при определенных предположениях (например, аппаратура работает корректно) ПО соответствует заданным свойствам. Ни на что другое доказательства не претендуют. Все же они

достаточно прочны, чтобы устранить необходимость выполнения некоторых тестов. Обычно нет необходимости в тестировании корректности свойств, доказанных математически.

- Ограничение в том, что пока эта техника на грани искусства, она требует специальной организации процесса разработки, специально обученной команды разработчиков. Помимо всего, обычно предполагается, что доказываемая часть программы создается на существенно урезанном языке программирования, когда приходится отказываться от жизненно важных средств: классов, наследования и его следствий (полиморфизм, динамическое связывание), универсальности, динамического создания объектов, рекурсии ...

Все эти свойства современной технологии облегчают конструирование больших программ с элегантной архитектурой, открытой для расширения и повторного использования, предоставляя программистам выразительную мощь языка программирования. В результате формальные методы пока применяются в той области, где существует один критерий – корректность. Сюда относятся жизненно важные системы, такие как системы управления самолетами или поездами, где должно быть сделано все, чтобы избежать ошибок функционирования. Пример ПО «Аэробуса» является показательным.

Остальная часть индустрии обычно не готова принять такой вид аскетизма, который требует техника доказательств от своих последователей. Ведутся серьезные исследования, чтобы сделать их более применимыми в главных направлениях разработки ПО. Тони Хоар выступил с инициативой «Grand Challenge» («Большой вызов»), предполагающей сосредоточенные международные усилия по созданию верифицируемого ПО. Можно, в самом деле, надеяться, что в ближайшие годы формальные методы покажут свои преимущества даже тем из нас, чьи программы при неправильном функционировании не угрожают жизни людей.

19.8. Модели способностей и зрелости

Наша последняя тема этой главы посвящена общему организационному подходу, который применяется в компаниях последние годы. Он соответствует духу идей моделей жизненного цикла, обсуждаемых ранее в этой главе, но расширяет их рамки.

Предположим, что вашей организации необходимо заключить контракт на разработку ПО с некоторой программистской компанией. Пока нет продукта, который можно было бы оценить, так что оценке может подлежать только сам процесс разработки. Компания убеждает вас, что у нее все находится под контролем, но как это проверить?

В начале 90-х годов необходимость объективной оценки программистских фирм возникла у Министерства обороны Соединенных Штатов – крупнейшего заказчика ПО для своих нужд. Для решения этой задачи Институту программной инженерии Университета Карнеги-Меллона был сделан заказ на разработку модели, определяющей уровень зрелости программистских фирм. Разработанная ими модель получила название «Capability Maturity Model» (акроним CMM), а позже модель была расширена и стала называться CMMI-моделью – «интеграционная модель зрелости и способностей». Эта модель оказала существенное влияние на некоторые сегменты рынка ПО, в частности:

- на оборонные контракты МО США – ее первого заказчика;
- программистские компании Индии, которые увидели в получении внешних сертификатов CMM (CMMI) возможность независимого подтверждения их зрелости, что открывало им дорогу на мировые рынки.

Модель CMMI используется и вне этих сообществ. Доля организаций, работающих с оборонными заказами и сертифицированных по CMMI, неуклонно сокращается, в 2004 году она составляла 40%.[1]

> Некоторые компании в поисках процесса улучшения организации работ и квалификации предпочитают другие модели. Серия стандартов 9000 ISO также устанавливает международные стандарты качества. Стандарт SPIQE комбинирует элементы предыдущих двух стандартов. В этом обзоре рассматривается только CMMI.

Область действия CMMI

CMMI и другие модели исследуют только процессы. Они нейтральны к используемым технологиям, языкам и инструментарию. Все они оценивают, имеет ли организация набор четких процедур для каждого рода деятельности, применяются ли эти процедуры, контролируется ли их применение, измеряется ли эффект, прилагаются ли усилия по их улучшению. Вспоминая предыдущее наше обсуждение о качестве ПО, скажем, что модели следят за *факторами процесса*, особенно за последними пятью из нашего списка.

Процесс сертификации аналогичен проверке пилотом самолета перед вылетом контрольного списка: если с очередным элементом все в порядке, то движемся дальше, если нет – выполняем предписанные действия, такие как вызов службы сопровождения.

Из-за акцента на формальные процедуры в ущерб технологиям ряд специалистов критически относятся к моделям процессов, полагая, что они главным образом служат интересам менеджеров проекта, подкладывающих соломку на случай неуспеха проекта, – они делали все по инструкции. И в самом деле, известны случаи провала проектов в организациях с высоким уровнем сертификации CMMI или ISO. Но это классический пример, когда выполняется необходимое, но не достаточное условие. Программным проектам, особенно большим, необходимо высокое качество процессов, но этого недостаточно, им для успеха необходима также и выдающаяся технология.

Ключевым в CMMI является понятие оценивания. Организации, желающие установить свой уровень зрелости, могут сделать это, обратившись к официальным оценщикам, аккредитованным в Институте программной инженерии. В 2005 году Институт аккредитовал 179 оценщиков, главным образом организации, занимающиеся такого рода деятельностью. Организации, получившие сертификат об уровне их зрелости, могут опубликовать его и использовать в рекламных целях для повышения своей привлекательности.

CMMI-дисциплины

Как указывает буква «I» в акрониме (Integration), CMMI пошла дальше CMM для того, чтобы расширить сферу воздействия на модели, отличные от программных. Четыре дисциплины кроме инженерии программ включают также:

- инженерию системы. Эта концепция покрывает аспекты системы, не связанные с программированием. Представьте себе, что разрабатывается ПО для автомобиля, холодильника. Здесь возникают собственные процессы, включающие аппаратуру, ПО и другие аспекты;
- интегрируемый продукт и процесс разработки;
- наблюдение за поставщиками. Выбор, контроль и координация работ всех поставщиков, участвующих в проекте. Большие проекты часто включают многих сторонних поставщиков.

[1] Российские компании, работающие на оффшорном рынке, также сертифицируются по CMMI, например, компания VDI.

Организации, реализующие CMMI и желающие получить сертификат, могут выбирать любую из этих дисциплин в зависимости от их деятельности и потребностей.

Цели, практики и области процессов

Основа CMMI – определение целей и рекомендуемых практик.

- Целью является достижение желаемого свойства процесса. Например, каждый проект должен иметь хорошие требования, описывающие потребности пользователей. Из этого выводимы две цели: «Разработка требований заказчиков» и «Анализ и проверка правильности требований». Недостаточно просто создать требования, должна быть формальная процедура проверки их реализуемости и проверки того, что они удовлетворяют всех сопричастников.
- Практика – это метод, помогающий в достижении цели. Примерами являются «Задать определение требуемой функциональности» и «Проанализировать требования для установления баланса между потребностями сопричастников и ограничениями проекта».

Как показывают примеры, каждая практика должна быть связана с некоторой целью. Если использовать программистскую технологию, то цель – это спецификация, а практика – ее реализация (выполняемая людьми).

Цели и соответствующие им практики группируются в коллекции, называемые областью процесса. Приведенные примеры могут рассматриваться как часть области процесса «Разработка требований».

Термин «область» не является интуитивно понятным, так что просто следуйте определению – область, содержащая коллекцию целей и практик, поддерживающих эти цели.

Две модели

CMMI существует в двух вариантах – ступенчатом, или поэтапном, и непрерывном.

- В поэтапном варианте задается уровень зрелости организации в целом. Организация в процессе развития может подниматься на следующую ступеньку, получая более высокий уровень зрелости: «Наше подразделение достигло CMMI сертификации уровня 4!». Недостатком является игнорирование разницы между разными видами деятельности. Организация может быть хороша в процессе конструирования ПО, но слаба в вопросах задания требований. Поэтапный вариант является доминирующим.
- В непрерывном варианте независимо оценивается каждая область процессов, поэтому он обеспечивает большую гибкость.

Общим для обоих вариантов является понятие уровня оценивания. В поэтапном варианте организация получает оценку от 1 до 5 (как в школе). Чем выше оценка, тем выше уровень контроля процессов. В непрерывном варианте оценивается каждая область, здесь добавляется уровень 0, свидетельствующий о том, что организация не занимается этой конкретной областью процессов.

В поэтапном варианте каждый уровень характеризуется своим набором областей процессов. Вы достигаете соответствующего уровня, если отвечаете соответствующим целям и применяете нужные практики. Например, для достижения уровня 2 необходимо удовлетворять области процесса «Управление требованиями» и другим областям, перечисленным ниже. Кроме того, каждый уровень имеет родовую цель и связанное множество родовых практик. Для уровня 2 родовой целью является «Организация управляемого процесса», означающая, что в компании определён процесс разработки и принимаются меры по его соблюдению. Связанными родовыми практиками являются такие практики, как «Планирование процесса» и «Обеспечение ресурсами».

Как следствие этих концепций, цели и практики разделяются на две категории.

- **Родовые**: характеризующие CMMI-уровень, но не принадлежащие какой-либо области процесса.
- Те, что принадлежат области процесса, называемые **специфическими**.

Уровни оценивания

Дадим общую характеристику уровней для поэтапного варианта. Более точное определение дается ниже приводимой таблицей, которая идентифицирует в отдельности родовые и специфические цели. Как говорилось, выделяется пять уровней.

1. **Начальный**. Характеризует организацию, в которой определено или выполняется небольшое число процессов. Некоторые процессы успешны, некоторые нет, но причины неизвестны. Это напоминает сбор грибов в дождливый осенний день: под этим деревом много подосиновиков, под этим их нет, но почему? Для меня оба дерева выглядят одинаково. В разработке ПО такая ситуация известна как «героический подвиг» — успех во многом зависит от людей, их желания предпринять героические усилия. Это же говорит о плохо контролируемых обстоятельствах каждого проекта.
2. **Управляемый**. На этом уровне есть реальный процесс. Организация ведет политику, включающую определение процесса, планирование его выполнения, здесь следят за распределением ресурсов, определяют ответственность за выполнение планов, проводят мониторинг, обзоры и дают отчеты перед высшим руководством. Сопричастники определены и их интересы учитываются. Другими словами, процесс определен и тщательно прослеживается.
3. **Определенный**. Это управляемый процесс (начиная с этого уровня, предполагается, что предыдущий уровень выполняется) с более систематическими процедурами. Главное отличие от предыдущего уровня в том, что наряду с общностью появляется возможность настройки, учитывающая специфику конкретного проекта. У организации есть глобальная, но настраиваемая модель процесса, и процесс для каждого проекта проходит настройку.
4. **Количественно управляемый**. В добавление к предыдущим требованиям процесс интенсивно использует не только количественные оценки, такие как измерение стоимости, время разработки, качество надежности и сервиса, но и применяет методы статистического управления качеством, анализируя данные в глубину и используя результаты как часть процесса.
5. **Оптимизирующий**. Этот уровень добавляет обратную связь. Данные, собранные при выполнении проекта, позволяют непрерывно улучшать сам процесс через инновационные изменения.

Следующая таблица более точно описывает, что должно достигаться на каждом уровне (начиная со 2-го, поскольку по определению на уровне 1 нет точных требований).

Уровень	Имя	Области процессов
2	**Управляемый**	Управление требованиями Планирование проекта Мониторинг проекта и управления Управление соглашениями с поставщиками Измерения и анализ Страхование качества процесса и продукта Управление конфигурацией

3	**Определенный**	Разработка требований Технические решения Интеграция продукта Верификация Проверка правильности Выделение организационного процесса Определение организационного процесса Организационный тренинг Управление интегрированным проектом Управление рисками Интегрированная команда Интегрированное управление поставщиками Анализ решений и выводы Организационное окружение для интеграции
4	**Количественно управляемый**	Производительность организационного процесса Количественное управление проектом
5	**Оптимизирующий**	Организационные инновации и развертывание Анализ причин и выводы

CMMI определяет на каждом уровне точное множество целей и практик. Мы не будем заниматься этими деталями, надеясь, что вы разберетесь с ними сами, изучая литературу по CMMI, где вы сможете найти и метод, известный как персональный процесс разработки, который применяют индивидуальные разработчики, руководствуясь изложенными идеями.

19.9. Дальнейшее чтение

Carlo Ghezzi, Mehdi Jazayeri and Dino Mandrioli: Fundamentals of Software Engineering, 2nd Edition, Prentice-Hall, 2002.

Хорошо известный учебник по инженерии программ, дающий полное представление о предмете. Другими хорошими учебниками являются: S.L. Pfleeger, J. Atlee (3rd edition, Prentice Hall, 2005) и Roger Pressman (6th edition, McGraw Hill, 2005).

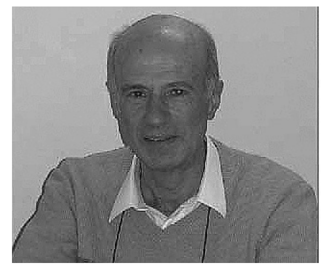


Рис. 19.7. Карло Чеззи (2008) Дино Мандриоли (2008)

IEEE Computer Society (Software Engineering Standards Committee): IEEE Recommended Practice for Software Requirements Specifications, IEEE Std 830-1998,

Доступна при условии регистрации на сайте:
ieeexplore.ieee.org/xpl/tocresult.jsp?isNumber =15571
Краткий стандарт, описывающий лучшие практики написания документов требований, включая рекомендуемую структуру документа, которая широко применятся в индустрии.

Bertrand Meyer: On Formalism in Specifications, in IEEE Software, vol. 3, no. 1, January 1985, pages 6-25.

Доступна на сайте: se.ethz.ch/~meyer/publications/computer/formalism.html.
Старая статья, объясняющая, почему полезно для задания спецификаций использовать математические методы.

John V. Guttag and James J. Horning: The Algebraic Specification of Abstract Data Types, in ActaInformatica, vol. 10, pages 27-52, 1978.

Конструктивная статья по теории абстрактных типов данных, лежащих в основе объектной технологии. Вводит понятие «достаточной полноты».

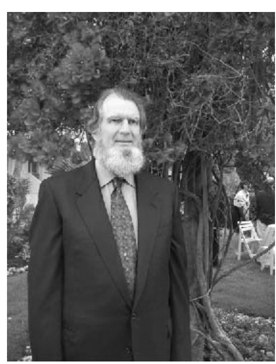


Рис. 19.8. Джим Хорнинг (2007)

KarlE. Wiegers: SoftwareRequirements, MicrosoftPress, 2003.

Набор полезных правил для написания хороших документов требований.

Michael Jackson: Software Requirements and Specifications: A Lexicon of Practice, Principles and Prejudices, ACM Press, Addison-Wesley, 1995,

Прекрасное обсуждение требований и спецификаций.

Axel van Lamsweerde: Requirements Engineering, Wiley, 2009.

Еще одна прекрасная книга по требованиям, наиболее современная, от одного из авторитетов в этой области. Хорошая теория и примеры.

Bertrand Meyer and Jim Woodcock (editors): VSTTE (Verified Software: Theories, Tools, Experiments), LNCS 4171, Springer-Verlag, 2008.

Proceedings of a 2005 conference at ETH Zurich,

Труды содержат работу Тони Хоара «GrandChallenge».
Хорошая оценка состояния искусства верификации программ.

Frederick P. Brooks: The Mythical Man-Month: Essays on Software Engineering, 20th Anniversary Edition, Addison-Wesley, 1995 (the original edition is from 1975, same publisher).

На русском языке: «Мифический человеко-месяц» см., например, на сайте:
www.webkomora.com.ua/ru/articles/web/management/man-month.html

Фред Брукс из IBM управлял разработкой OS/360, одной из первых сложных операционных систем, доступной на серии компьютеров. Эта книга, где он суммирует свой опыт в ко-

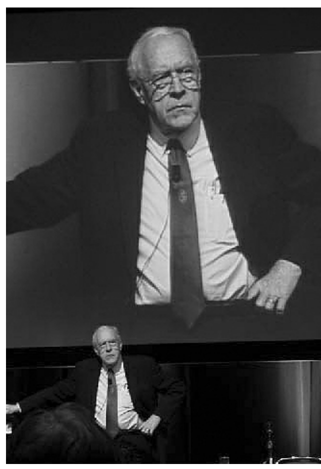


Рис. 19.9. Фредерик Брукс (2007)

ротких эссе, должна быть упомянута, так как считается классикой в инженерии программ, она в большой степени стала народным фольклором.

Software Engineering Institute: Capability Maturity Model Integration (CMMI)

Обзор, доступный на сайте:

www.sei.cmu.edu/cmmi/adoption/pdf/cmmi-overview07.pdf.

Software Engineering Institute: Capability Maturity Model® Integration (CMMISM), Version 1.1, CMMISM for Systems Engineering, Software Engineering, Integrated Product and Process Development and Supplier Sourcing (CMMI-SE/SW/IPPD/SS, V1.1) Staged Representation CMU/SEI-2002-TR-012 ESC-TR-2002-012.

Доступна на сайте:

tinyurl.com/kf9uy (shorthand for www.sei.cmu.edu/pub/documents/02.reports/pdf/02tr012.pdf#search=%22cmmi%20staged%20representation%22).

Это официальное, детальное описание CMMI, поэтапного представления.

Непрерывный вариант: tinyurl.com/gjla9

Watts S. Humphrey: PSP: A Self-Improvement Process for Software Engineering, Addison-Wesley, 2005

Рис. 19.10. Уотс Хэмпфри (2007)

Описывает персональный процесс разработки – PSP (Personal Software Process) для программистов, применяющих стиль правил практики инженерии, производных частично от идей CMMI об ответственности и воспроизводимости.

19.10. Ключевые концепции, изложенные в этой главе

- Инженерия программ включает в себя программирование наряду с другими видами деятельности – техническими и не техническими, необходимыми при создании программных систем. Ее целью является создание индустриальных программных продуктов с заданными стандартами качества.
- Инженерия программ включает пять главных категорий задач, охватываемых акронимом DIAMO: описание, реализацию, оценку, управление, функционирование.
- Инженерия программ воздействует как на процесс разработки, так и на конечный продукт.
- Качество продукта и процесса включает много факторов, начиная от корректности и эффективности до стоимости и воспроизводимости.
- Программный проект должен иметь четкое представление о сопричастниках – лицах, заинтересованных в проекте, и о том, какие цели важны для каждой их категории.
- Разработка ПО включает несколько определенных задач, которые модели жизненного цикла пытаются выстроить последовательно. Agile-методы (гибкая методология) уделяют меньше внимания процессам, отдавая предпочтение работающему коду и человеческому взаимодействию.
- Анализ требований к системе является основной задачей каждого проекта. Он включает точное описание свойств системы, не содержащее предположений об их возможной реализации, точное описание потребностей сопричастников, что позволяет следить за реализацией системы.
- Требования к системе включают функциональные аспекты, специфицирующие функции системы, и не функциональные аспекты, такие как ограничения производительности. Существует IEEE-стандарт для структурирования документов требований.
- Проблемные свойства отражают внешние ограничения, машинные – выражают решения о свойствах программной системы.
- Верификация и проверка правильности могут использовать динамические методы, в частности тестирование, и статические, такие как обзоры кода и проекта, статический анализ, доказательства корректности и проверки на моделях.
- Целью тестирования является обнаружение дефектов системы, выявляемых при отказах во время выполнения тестов.
- Модель CMMI определяет пять уровней зрелости процесса разработки, применяемого в организации. На пятом, самом высоком уровне процесс определен, документирован, подвержен измерениям, воспроизводим и самоулучшаем.

Новый словарь

Adequacy	Адекватность	**Built-in assessment**	Встроенное оценивание
Correctness	Корректность	**Correctibility**	Способность к изменениям
Cost control	Управление стоимостью	**Efficiency**	Эффективность
Extendibility	Расширяемость	**Factor (of software quality)**	Фактор (качества ПО)
Goal (CMMI)	Цель (CMMI)		
Lifecycle	Жизненный цикл	**Maintenance**	Сопровождение
Measurability	Измеримость	**Portability**	Переносимость
Practice (CMMI)	Практика (CMMI)	**Predictability**	Предсказуемость

Process (vs product)	Процесс (в сравнении с продуктом)	**Process area (CMMI)**	Область процесса (CMMI)
Product (vs process)	Продукт (в сравнении с процессом)	**Production software**	Производство ПО
Reproducibility	Воспроизводимость	**Reusability**	Повторное использование
Robustness	Устойчивость		
Security	Безопасность	**Self-improvement**	Самоулучшение
Software engineering	Инженерия программ	**Stakeholder**	Сопричастник

19-У. Упражнения

19-У.1. Словарь

Дайте точное определение терминам словаря.

19-У.2. Сопричастники

Могут ли сопричастники программного проекта быть соперниками? Обсудите, в какой части они или концепции о них могут играть роль в построении ПО и управлении проектом.

19-У.3. Лучше позже или лучше с ошибками, но раньше?

В обзоре CMMI, указанном в разделе «Дальнейшее чтение», приводится высказывание (и его критика) неназванного старшего менеджера: «Я бы предпочел вовремя выпустить проект с ошибками, чем опоздать с выпуском. Позже мы всегда сможем исправить ошибки». Обсудите это высказывание с позиций инженерии программ.

Часть VI

Приложения

Эта часть дополняет главные материалы книги, представляя введение в четыре важных языка программирования: Java, C#, C++ и (кратко) C.
Описания языков ориентированы на читателей этой книги. Нотация и концепции сравниваются с уже знакомой нотацией и концепциями Eiffel.
Описания не покрывают языки во всей их полноте, но включают достаточно материала, чтобы можно было начать писать программы на этих языках, если освоены концепции этой книги. Для практического использования необходимо, конечно, знакомство с окружением (компилятором, интерпретатором, средой разработки), так что понадобятся соответствующие материалы, доступные в книгах или в режиме онлайн.
Описание первых трех языков дается независимо, поскольку не каждый читатель захочет знакомиться со всеми. Языки Java, C# and C++ разделяют многие характеристики (первые два находятся под влиянием C++, а он сам – под влиянием C). Это означает, что если читать все три описания, то неизбежно встретятся повторы. Четвертое приложение дает краткий обзор языка C как подмножества языка C++, предполагая, что приложение по C++ уже прочитано.
Последнее приложение представляет базисную информацию об окружении EiffelStudio, полезную при запуске примеров этой книги и работе с системой Traffic.

Введение в Java (по материалам Марко Пиккони)

A.1. Основы языка и стиль

Язык Java появился в 1995 году в результате внутреннего исследовательского проекта Sun Microsystems, руководимого Джеймсом Гослингом (ключевой вклад в разработку языка внесли также Билл Джой, Гей Стил и Джилард Брачча).

Язык появился в нужное время, отвечая на возникшие в тот период потребности.

- После начального энтузиазма, вызванного появлением в конце восьмидесятых годов языка C++ и объектной технологии, широкое недовольство стали вызывать сложность языка и его «гибридный» подход, сочетающий совместимость с необъектным языком C.

Рис. А.1. Джеймс Гослинг (2007)

- Широкое распространение Интернета и всемирной паутины – World-Wide Web – явно требовало универсального механизма безопасного выполнения программ в браузере.

Проект Java вначале предполагался для создания апплетов – модулей для сетевых приложений. Как отмечалось ранее, апплеты не стали доминирующей моделью, как провозглашалось изначально, но использование Java быстро расширилось на многие другие области.

Следующие свойства характеризуют модель программирования Java.

- Тесная связь между языком программирования и платформой, которая основана на виртуальной машине, называемой JVM (Java Virtual Machine).
- Акцент на переносимость, отражаемый в лозунге: «Раз напишешь, всюду выполнишь». Компилятор Java транслирует Java-программу в байт-код JVM, который затем на многих платформах интерпретируется или компилируется в машинный код.
- Синтаксис, общий стиль языка и базисные операторы заимствованы из семейства языков C – C++.
- Строго типизированная ОО-модель, которая включает многие механизмы, изучаемые в этой книге: классы, наследование, полиморфизм, динамическое связывание, универсальность (добавленная в последующих версиях). Некоторыми опущенными элементами являются множественное наследование (допускается множественное наследование интерфейсов, как мы увидим), контракты и агенты. ОО-часть системы типов не включает примитивные типы.
- Помимо того, что предлагает язык, разработку поддерживает множество библиотек, ориентированных на различные области приложения.

А.2. Общая структура программы

Прежде чем рассмотреть общую структуру Java программ, начнем с обзора виртуальной машины Java.

Виртуальная машина Java – JVM

Виртуальная машина Java – это программная система, обеспечивающая механизмы поддержки выполнения Java программ. В зависимости от контекста JVM можно рассматривать и как общую спецификацию этих механизмов, и как конкретную реализацию. Принципиальными механизмами являются:

- загрузчик классов, который управляет классами и библиотеками файловой системы, динамически загружая классы в формате байт-кода;
- верификатор, который проверяет, чтобы байт-код удовлетворял фундаментальным ограничениям надежности и безопасности: безопасности типов (не null ссылки всегда ведут к объектам ожидаемых типов); скрытия информации (доступ к компоненту отвечает правилам видимости); правильности ветвления (ветви всегда должны вести к правильному местоположению); инициализации (каждый элемент данных инициализирован перед его использованием);
- интерпретатор, программный эквивалент ЦПУ (процессора) физического компьютера, – выполняет байт-код;
- компилятор, работающий «на лету» (называемый джиттером – JIT – Just In Time компилятором), транслирует байт-код в машинный код для данной конкретной платформы, выполняя различные оптимизации. Наиболее широко используется JIT-компилятор «Hot Spot» фирмы Sun.

Пакеты

Программы Java состоят из классов, как и в других ОО-языках. Но Java предлагает модульную структуру, стоящую над уровнем классов – пакеты. Пакет – это группа классов (подобно кластеру Eiffel).

Пакеты выполняют три главные роли. Первая – помощь в структурировании программной системы и библиотек. Пакеты могут быть вложенными и, следовательно, дают возможность организовать классы в иерархическую структуру. Вложенность структуры является концептуальной, но не текстуальной. Другими словами, вы не объявляете пакет как таковой (с входящими в него классами), вместо этого при объявлении класса указываете имя пакета, если таковой имеется.

```
package p;
class A {… Объявление компонентов (членов) A …}
class B {… Объявление компонентов (членов) B …}
… Объявление других классов …
```

Если это содержимое исходного файла, то все заданные классы принадлежат пакету p. Директива package, если присутствует, должна быть первой строкой файла. Вложенные пакеты используют нотацию с точкой: p.q означает, что q – подпакет p.

Директива package имеет статус «возможна», в ее отсутствии все классы в файле будут рассматриваться как принадлежащие специальному пакету по умолчанию.

Вторая роль пакетов в том, что они являются единицей компиляции. Вместо индивидуальной компиляции классов, можно скомпилировать весь пакет в единый архив – «Java Archive» (JAR- файл).

В своей третьей роли пакеты обеспечивают механизм «пространства имен», что позволяет разрешать конфликты совпадающих имен разных классов, приходящих, например, из библиотек, поставляемых разными провайдерами. При ссылке на класс A, принадлежащий пакету p, всегда можно использовать полное квалифицирующее имя: p.A. Эта техника применима и к подпакетам, как в p.q.r.Z. Чтобы избежать полной квалификации, можно использовать директиву *import*, написав:

```
import p.q.*;
```

Это позволяет в остальной части файла применять классы из p.q без квалификации до тех пор, пока не возникают конфликты (символ * в директиве означает «все классы из пакета», не включая подпакеты). Для разрешения неопределенностей доступна полная квалификация.

Механизм пакетов приходит с некоторыми методологическими рекомендациями. Первая – использовать явную форму задания пакета, включая каждый класс в именованный пакет, не применяя пакеты по умолчанию. Другая рекомендация связана с тем, что пакеты и имена пространств только перемещают проблему конфликтов имен на более высокий уровень, поскольку могут конфликтовать и имена пакетов. Для минимизации таких ситуаций предлагается стандартное соотношение для именования пакетов, когда в имя пакета включается уникальное имя сайта соответствующей организации, перечисляя компоненты в обратном порядке. Например, для пакета, создаваемого нашей группой (имя домена se.ethz.ch), имя может быть следующим:

```
ch.ethz.se.java.webtools.gui
```

Выполнение программы

Чтобы запустить из командной строки Java программу на выполнение, нужно выполнить команду:

```
java C arg1 arg2 …
```

Здесь *C* – это имя класса, а возможные аргументы *arg1 arg2 …* являются строками. В классе *C* должен находиться метод с фиксированным именем *main*, который и будет запущен на выполнение:

```
public static void main(String[] args) {
    … Код метода main …
}
```

В отличие от Eiffel при запуске не создается объект, так как статическому методу (как объясняется ниже) не требуется объект. Конечно же, main в своей работе обычно создает объекты или вызывает другие методы, создающие объекты. Возможный формальный аргумент является массивом строк (String[]), соответствующий приведенному выше вызову с arg1 arg2 … Квалификатор public, также изучаемый ниже, делает метод main доступным всем клиентам[1].

А.3. Базисная ОО модель

Рассмотрим теперь основные ОО-механизмы Java. Обсуждение предполагает знакомство с предыдущими главами.

Система типов Java

Большинство типов, определяемых Java-программистом, будут, как в большинстве примеров этой книги, ссылочными типами, каждый основанный на классе. На вершине иерархии

[1] Обычно под клиентом понимается клиентский класс. Но таким клиентам main не доступна, поскольку она может быть вызвана только извне программной системы, задавая точку «большого взрыва».

классов находится класс Object – прародитель всех классов, являющийся аналогом класса *ANY* в Eiffel (но здесь нет аналога класса *NONE*).

В отличие от системы типов Eiffel и C#, изучаемого в следующем приложении, не все типы Java построены на классах. Простые типы, встроенные в язык, называемые примитивными типами, не входят в ОО-систему типов, – переменные этих типов не рассматриваются как объекты. Здесь Java следует концепциям языка C++. Примитивными типами Java являются следующие типы:

- **boolean**, логический тип;
- **char**, тип, представляющий символы Unicode (16-бит);
- целочисленные арифметические типы: **byte, short, int и long**, соответственно представляющие целые (8-бит, 16-бит, 32-бит и 64-бит);
- типы с плавающей точкой для вещественных чисел: **float** (32-бит) и **double** (64-бит).

Нельзя непосредственно использовать значения этих типов как объекты, например, в структуре данных, хранящей объекты произвольных типов. Но можно выполнить обертку, обернув значения в одежды классов и превратив их в объекты. Возможна и обратная операция. Эти операции называются «boxing» и «unboxing» (помещение значения в ящик объекта и извлечение из ящика). Java предоставляет соответствующее множество обертывающих классов: **Boolean**, **Character**, **Byte**, **Short**, **Integer**, **Long**, **Float**, **Double** (язык чувствителен к регистру, так что **Byte** отличается от **byte**). Объявим:

```
int i;                  // Примитив
Integer oi;             // Обертка
Возможны взаимные присваивания:
oi=i;         // Сокращение записи oi = Integer.valueOf(i);
i = oi;           // Сокращение записи i = oi.intValue()
```

Как показано в комментарии, присваивание требует вызова функций преобразования между примитивами и их объектными двойниками, но явно нет необходимости вызова этих функций, все делается автоматически, «за кулисами».

Выражение oi.intValue() иллюстрирует еще одну разницу с концепциями этой книги: Java не применяет принцип унифицированного доступа. Функция без аргументов intValue из класса **Integer**, вызываемая с пустым списком аргументов, как выше, синтаксически явно отличается от атрибута.

Классы и члены

Класс содержит члены (members), термин Java для компонентов (features) класса. Член может быть полем (атрибутом), методом или конструктором (процедурой создания). Текст класса может также содержать инициализатор: анонимный блок кода, вызываемый в момент инициализации. Следующий текст класса содержит примеры всех этих категорий:

```
class D {
    String s;                 // Поле переменной
    final int MAX = 7;        // Поле константы
    T func (T1 a1, T2 a2){
           // Метод (функция) с двумя аргументами типа T1 и T2,
           // возвращающий значение типа T.
        … Код функции func …
    }
```

```
    void proc(){
          // Метод (процедура) без аргументов.
       … Код proc …
    }
    D(){
       // Конструктор без аргументов: Имя конструктора совпадает с именем класса
          // Конструктор не возвращает значения. Код конструктора …
    }
    D (T1 a1){
          // Еще один конструктор с одним аргументом.
       … Код конструктора …
    }
    {
          // Инициализатор
       … Код инициализатора …
    }
}
```

Скрытие информации

У члена класса есть статус экспорта, который может принимать одно из четырех значений, перечисленных здесь в порядке уменьшения доступности:

- **public**: доступен любому клиенту;
- **protected**: доступен потомкам и классам пакета;
- **package** (не является ключевым словом, но является статусом по умолчанию): доступен классам пакета;
- **private**: доступен только самому классу.

Эти квалификаторы также применимы к классам, в частности, по той причине, что классы Java могут быть вложенными. Для класса верхнего уровня, не вложенного в другой класс, единственно возможными значениями могут быть значение по умолчанию (класс доступен в своем пакете) и значение **public.**

Из-за отсутствия поддержки унифицированного доступа смысл статуса доступа отличается от того, что мы видели в этой книге. Экспорт члена класса, являющегося полем (для первых трех значений статуса), дает соответствующим клиентам право как на чтение, так и на запись значения поля. Это означает, что можно получать непосредственный доступ к полям удаленных объектов.

```
x.a = b;
```

Это противоречит принципу скрытия информации и ведет к общей методологической практике никогда не экспортировать поля, придавая им статус private, и при необходимости поставлять их с геттер-функцией и сеттер-процедурой.

Статические члены

Еще одной концепцией Java, отличающей ее от строго ОО-стиля, используемого в этой книге, является поддержка статических методов.

Для доступа к члену класса нужен целевой объект. Обычная конструкция доступа, характерная для ОО-стиля, – target.member (возможно, с аргументами при вызове метода), где tar-

get обозначает объект. Если цель явно не указывается, то целевым является текущий объект. Текущий объект в Java имеет имя **this** (**Current** в Eiffel).

В языке Java возможно объявлять статические члены, которым не требуется целевой объект, они вызываются как C.member, где C – имя класса. Статическому методу класса доступны только статические поля и статические методы этого класса (нестатические члены недоступны, так как они требовали бы целевого объекта).

Главная программа, *main*, должна быть статической, как отмечалось выше, причина в том же: не существует объекта, вызывающего этот метод (в отличие от Eiffel, где выполнение состоит в создании объекта и вызова процедуры создания на нем)[1].

Абстрактные классы и интерфейсы

Можно отметить метод как абстрактный, указав тем самым, что реализация будет обеспечена потомком. Класс, в котором появляется хотя бы один абстрактный метод, также должен быть помечен как абстрактный:

```
public abstract class Vehicle {
    public abstract void load (int passengers);         // Нет тела метода.
    … Описание других методов абстрактных или не абстрактных (эффективных в
    … терминологии Eiffel) …
}
```

Это соответствует отложенным **deferred**-методам и классам Eiffel без возможности задания контрактов для абстрактных методов.

Еще одно отличие от механизма отложенных классов состоит в том, что абстрактные классы, подобно другим Java-классам, как мы увидим при рассмотрении наследования, могут участвовать только в одиночном наследовании. Класс может наследовать максимум от одного класса, абстрактного или нет. Поэтому невозможныо, используя только классы, комбинировать две или более абстракции. Для ослабления этого ограничения Java обеспечивает еще одну форму абстрактного модуля – **interface**. Такой класс является эквивалентом абстрактного класса, чьи методы все абстрактны (константы и вложенные типы допускаются). Объявление интерфейса выглядит следующим образом:

---

[1] Предлагаемая интерпретация статических элементов в языках Java и C# не единственна. Возможна другая интерпретация. В языке C# есть статический конструктор, в Java – статический инициализатор. Статический конструктор создает статический объект, существующий в единственном экземпляре, дает ему имя, совпадающее с именем класса. Этот объект, содержащий набор статических полей класса, и является целевым объектом при вызове статических полей и статических методов класса. Поэтому вызов C.member осуществляется в том же объектном стиле, что и вызов target.member.
Статический конструктор класса существует по умолчанию, выполняя важную работу. Если у класса есть константы примитивных типов, то они являются полями создаваемого объекта, а не полями динамических объектов, так как нужны в единственном экземпляре. Если классу нужны собственные константы (как мнимая единица для класса COMPLEX, задающего комплексные числа), то статический конструктор (инициализатор Java) – то место, где такая константа создается. Если классу нужны однократные методы (аналог once методов Eiffel), то они становятся статическими методами класса. Программист может добавить собственный код в конструктор, хотя он не может вызывать статический конструктор (инициализатор). Этот вызов делается автоматически с гарантией, что он выполняется до начала работы с объектами класса.
Аналогичная интерпретация имеет место и при вызове метода main.

```
interface I {
        // Константы
    int MAX = 4;
        // Абстрактные методы
    void m1(T1 a1);
    String m2();
}
```

Заметьте, что объявление только специфицирует имена и сигнатуры, а также значения для констант. Все методы интерфейса автоматически квалифицируются как **abstract** и **public**, а все атрибуты – **public**- и **static**-константы.

Классы могут реализовать один или несколько интерфейсов, как в данном примере:

```
class E implements I, J{
    void m1(T1 a1) { … Реализация m1… }
    String m2(){ …Реализация m2}
    … Реализация методов интерфейса J (предполагается другой интерфейс)…
    … Другие члены E …
}
```

Перегрузка

Для классов Java допускается существование методов с одним именем при условии, что их *сигнатуры отличаются* (либо они имеют разное число аргументов, либо, при совпадении числа аргументов, – разный набор типов). Эта концепция известна как *перегрузка* методов.

Перегрузка используется при создании объектов, поскольку конструкторы, которых, как правило, у класса много, имеют одно имя (имя класса) и отличаются только сигнатурой.

Помимо конструкторов, в других ситуациях перегрузку лучше не применять, – в пределах класса разные вещи должны иметь разные имена. Кроме того, в языках, поддерживающих наследование, перегрузка смешивается с переопределением.

Модель периода выполнения, создание объектов и инициализация

Модель периода выполнения Java подобна модели, обсуждаемой в этой книге, в частности, Java предполагает автоматическую сборку мусора.

Ссылка, не присоединенная к любому объекту, имеет значение *null*, которое является значением по умолчанию для ссылочных переменных.

Ключевое слово void используется в других целях – оно указывается в качестве возвращаемого значения для методов, являющихся процедурами, сигнализируя, что процедуры не возвращают результата, в отличие от функций.

Программы создают объекты динамически, используя конструкцию **new**, как в примере:

```
o = new D (arg1);       // Ссылка на класс D из ранее приведенного примера,
                               // в частности, на его второй конструктор.
```

Здесь *o* типа *D*. Часто объявление комбинируется с созданием объекта, не отделяя, как в Eiffel, объявление (статику) и операторы (динамику):

```
D o = new D (arg1);
```

В отличие от **create** Eiffel в конструкции new необходимо повторять имя класса.

Выражение new ссылается на один из конструкторов класса. Как отмечалось, конструкторы не имеют собственных имен (как это делают методы класса), но все используют имя класса, создавая перегрузку. Класс D, приведенный ранее, имеет два конструктора, один без аргументов, другой – с одним аргументом типа T1 или потомка T1 (в противном случае вызов конструктора приведет к возникновению ошибки).

Перегрузка имеет свои ограничения. Например, для класса, задающего точку на плоскости, хочется иметь два конструктора, которым передаются координаты точки соответственно в декартовой или полярной системе координат. Но у этих различных конструкторов сигнатуры совпадают, поэтому приходится одному из них добавлять искусственный аргумент.

Класс может не объявлять ни одного конструктора; в этом случае в класс добавляется «конструктор по умолчанию» – без аргументов и с пустым телом.

Процесс создания объекта сложен. Полный эффект от вызова конструктора, такого, как был указан выше, состоит в выполнении следующей последовательности.

I1 Выделить память объекту типа D.

I2 Рекурсивно выполнить шаги I3 – I8 по отношению к родителям D (в данном случае у D нет явных родителей, хотя есть неявный родитель Object. Если же были бы родители, то рекурсивно шаги выполнялись бы для них, подымаясь до Object).

I3 Для всех статических полей установить значения по умолчанию.

I4 Если при объявлении у некоторых статических полей заданы значения, то установить их (как в **static int** n = 5;).

I5 Выполнить все статические блоки инициализатора.

I6 Для всех нестатических полей установить значения по умолчанию.

I7 Если при объявлении у нестатических полей заданы значения, то установить их.

I8 Выполнить все нестатические блоки инициализатора.

I9 Вызвать конструктор родителя.

I10 Выполнить тело конструктора.

Шаг I9 отражает правило Java, гласящее, что каждый конструктор должен вызывать конструктор родителя. Правило рекурсивно, поэтому цепочка вызовов поднимается вплоть до вызова конструкта класса Object. Выбор родительского конструктора производится следующим образом.

- Текст конструктора в качестве первого оператора может задавать вызов специального метода super, предавая ему при необходимости аргументы. Ключевое слово super означает родителя, так что, по сути, вызывается конструктор родителя. Учитывая перегрузку, состав переданных аргументов позволяет однозначно выбрать нужный конструктор.
- Если вызов super явно не задан, то это означает вызов конструктора без аргументов – super(); в этом случае родитель должен иметь такой конструктор, заданный явно или по умолчанию.

Причины этих правил не ясны. Намерение, наверное, состояло в том, чтобы экземпляр потомка удовлетворял ограничениям согласованности, заданными предками. Механизм цепочки конструкторов может быть попыткой в достижении такой согла-

сованности в отсутствии понятия инварианта класса, позволяя явно выразить ограничения[1].

Инициализация полей на шагах I3 – I6 использует значения по умолчанию, как в Eiffel. В отличие от Eiffel, правила применимы только к полям; локальные переменные должны инициализироваться вручную. Компилятор выдает предупреждение, если их инициализация не выполнена.

Массивы

Массивы Java являются объектами, распределяемыми динамически, как это делалось в этой книге. Для определения массива используется объявление, такое как:

```
int[] arr;                              // Массив целых
```

Для создания объекта, представляющего массив, используется, как обычно, конструкция **new**:

```
arr = new int[size];
```

Здесь *size* – целочисленное выражение (не обязательно константа). В отличие от Eiffel, массивы не изменяют размеры. Доступ к элементам массива использует нотацию с квадратными скобками, как arr[*i*]. Следует помнить, что нижняя граница индексов фиксирована и равна 0, так что в вышеприведенном примере индексы меняются от 0 до *size* – 1. Элементу массива можно присвоить значение, например, так:

```
arr[i] = n;
```

Выражение arr.length (length – поле только для чтения) определяет число элементов массива. После вышеприведенного создания массива его значение будет *size* + 1 (так как *size* определяет значение верхней границы индекса, а нижняя равна 0). Типичная итерация по массиву, использующая цикл, детали которого будут приведены ниже, имеет вид:

```
for (int i=0; i < arr.length ; i++)
    {… Операции над arr[i] …}
```

Здесь i++ увеличивает целое i на 1. Заметьте, что условие продолжения цикла отражает тот факт, что максимальный допустимый индекс равен arr.length – 1.

Можно иметь многомерные массивы как массивы массивов, например:

```
int[][][] arr3; //Трехмерный массив
arr3[i][j][k]
```

[1] Объяснение может быть более простое. Конструкторы – это единственные методы класса, которые потомок не наследует. Но он их вызывает. Работа по созданию объекта, как говорилось, начинается с создания цепочки вызовов конструкторов, на вершине которой стоит конструктор прародителя – Object. Каждый конструктор в цепочке выполняет свою часть работы по созданию объекта. Работают конструкторы в обратном порядке. Первым начинает конструктор прародителя, создавая праобъект. Затем следующий конструктор модифицирует этот объект, добавляя свои поля и выполняя другие нужные действия. Последним, завершая отделку объекта, работает вызванный конструктор, инициировавший создание объекта.

Обработка исключений

Исключение – это событие, происходящее во время выполнения программы, когда нормальное ее выполнение прерывается и не может быть продолжено без специальной обработки. Типичные причины события включают: вызовы с **null**-ссылками (void-вызов: x.f , где *x* имеет значение **null**), деление целого на нуль. В Java разработчик может явно *включить исключение*, когда в результате анализа проблемной ситуации становится ясно, что нормально продолжать работу данный алгоритм не может. Включение (выбрасывание) исключения делается так:

```
throw e1;
```

Здесь e1 – тип исключения, который должен быть потомком библиотечного класса *Throwable*. Точнее, тип исключения в большинстве случаев является потомком класса Exception, что справедливо и для типов исключений, определяемых программистом. Класс Exception является наследником класса Throwable. Другим наследником является класс Error, покрывающий системные ошибки периода выполнения. Программа Java способна обрабатывать возникающие исключения в следующем стиле:

```
try {
    … Нормальные операторы, во время выполнения которых потенциально может
     …возникнуть исключение …
} catch (ET1 e) {
    … Обработка исключения типа ET1, детали исключения в объекте e …
} catch (ET2 e) {
    … Обработка исключения типа ET1, детали исключения в объекте e …
}… Другие возможные случаи …
finally {
    … Финальная обработка, независимо от того, было исключение или нет …
}
```

Если в **try**-блоке выбрасывается исключение одного из перечисленных типов, здесь ET1*,* ET2 ..., то выполнение, прерванное в точке возникновения исключения, не будет продолжать работу **try**-блока, но продолжится в соответствующем **catch**-блоке. Блок **finally**, если он задан, выполняется во всех случаях: его типичное назначение – освобождение ресурсов, например, закрытие открытых файлов.

При любом появлении исключения автоматически создается объект исключения – экземпляр подходящего потомка Throwable. Программа обработки может получить доступ к этому объекту в соответствующем **catch**-блоке (во всех примерах *e* – имя этого объекта). У объекта, задающего исключение, есть свойства, такие как состояние стека вызовов, имя исключения в виде строки текста и другие.

Если встретилось исключение, чей тип не совпадает с типами, перечисленными в **catch**-блоках, или исключение встретилось вне охраняемого **try**-блока, то оно передается вверх по цепочке вызовов – вызывающему методу. Здесь опять-таки рекурсивно исключение может быть обработано, если предусмотрен **catch**-блок, или передано наверх. При завершении цепочки вызовов, если обработка исключения не выполнена, то программа завершается стандартным сообщением об ошибке – обрабатывается стандартным **catch**-обработчиком исключения.

В языке Java вводится интересное разграничение на «проверяемые» и «не проверяемые» – «checked» и «unchecked» – исключения. Положение в иерархии типов исключения с вершиной Throwable определяет, какие исключения являются проверяемыми, как показано на следующем рисунке:

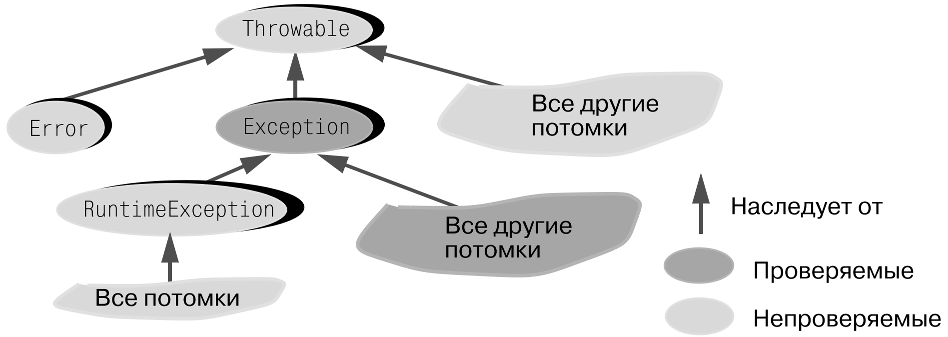


Рис. А.2. Проверяемые и непроверяемые классы исключения Java

Проверяемые исключения обеспечивают механизм, подобный контрактам: правило говорит, что если метод может выбрасывать проверяемое исключение, он должен объявить его, и тогда все клиенты, вызывающие метод, обязаны предусмотреть обработку исключения. Для указания того, что метод может выбрасывать исключение, используется ключевое слово **trows** (не путайте с оператором **throw**, который выбрасывает (включает) исключение):

```
public r(…)throws ET1, ET2{
    … Код r, включающий операторы
              throw e1;    // Для e1 типа ET1
              throw e2;    // Для e2 типа ET2
…
}
```

Если *r* включает **throw** e3; для e3 проверяемого типа ET3, и e3 не появилось в предложении **throws**, метод признается ошибочным – если только его тело не содержит **try**-блок с ветвью в форме **catch** (ET3 e), гарантирующий, что исключение будет обработано внутри метода, а не передано вызывающему методу.

Для вышеприведенного объявления любой вызов *r* из некоторого метода должен находиться в охраняемом блоке, сопровождаемом **catch**-блоками перечисленных типов, здесь ET1 и ET2.

Этот тщательно спроектированный механизм имеет привлекательные стороны, но и некоторые изъяны. Ограничение в том, что так можно заставить использовать **throws**-спецификации только для исключений, определенных программистом, в то время как большинство исключительных ситуаций связано с появлением системных исключений (void-вызовы и так далее). Когда правило заставляет программиста использовать **try**-блок, для ленивого программиста проще всего написать заглушку – ничего не делающий **catch**-блок, тем самым дискредитируя цель механизма. Вероятно, по этой причине в языке C# механизм исключений, практически идентичный механизму Java, не включает проверяемые исключения. Все

же проверяемые исключения способствуют разумной дисциплине в обработке исключений, и их следует использовать для исключений, создаваемых программистом.

А.4. Наследование и универсальность

В исходном варианте Java присутствовало только единичное наследование и отсутствовала универсальность. Позже в язык была добавлена универсальность, но наследование так и осталось единичным, если не считать множественного наследования интерфейсов.

Наследование

Для указания того, что один класс наследует от другого, используется ключевое слово **extends**. Класс, наследующий от интерфейса, использует другое ключевое слово – **implements**. Оба варианта могут комбинироваться, но первым указывается **extends**:

```
public class F extends E implements I, J {…}
```

Класс без **extends** рассматривается как наследник **Object –** предка всех классов.
Класс можно объявить как законченный – final, что запрещает наследование от него:

```
final class M …
```

У Java нет механизма переименования для разрешения конфликтов имен. Если два метода наследуются от класса и от интерфейса и их имена совпадают, но сигнатуры отличаются, то конфликта нет, поскольку в этом случае имеет место обычная для Java перегрузка. Если же у методов с совпадающими именами (они могут приходить и от двух разных интерфейсов) и сигнатуры совпадают, то методы конфликтуют, и нет простого способа разрешения конфликта.

Переопределение

Метод родителя можно переопределить у потомка. Такой метод не может быть статическим, и переопределение сохраняет сигнатуру. Если сигнатура не соблюдается, то это обычная перегрузка, а не переопределение. Здесь требуется особое внимание, так как оба механизма выполняются по умолчанию и не используют специальных ключевых слов. Просто объявляется новый член класса с тем же именем, что у родителя, и в зависимости от того, сохраняется сигнатура или нет, говорим о перегрузке, о переопределении или об ошибке, когда в классе появляются два метода с одним именем и одной сигнатурой.

Возвращаемый тип не является частью сигнатуры и не играет роли в правилах перегрузки. Для переопределяемого метода он обычно совпадает с типом оригинала, но он может быть и потомком типа оригинала. Такая ситуация известна как *ковариантное* переопределение (Eiffel предполагает ковариантное переопределение как для результата, так и для аргументов метода, что приводит к определённым проблемам системы типов).

Эквивалентом механизма **Precursor** для доступа к оригинальной версии переопределенного метода является конструкция **super**, уже встречающаяся при работе с конструкторами. Например:

```
public display(Message m) {       // Переопределение родительского метода display
    super(m);                     // Выполнение метода display, заданного родителем
    … Другие операции, расширяющие работу родителя …
}
```

Для полей (атрибутов) использование того же имени у потомка перекрывает оригинальную версию[1].

Переопределение члена может расширить статус видимости, но не ограничить его.

Полиморфизм, Динамическое связывание и Кастинг

Полиморфизм и динамическое связывание являются политикой умолчания, как в этой книге. Другими словами, если e1 типа E, f1 типа F, и F – потомок E, то можно использовать полиморфное присваивание:

```
e1 = f1;
```

После присваивания вызов в форме e1.r () будет использовать F версию r, если F переопределил метод r.

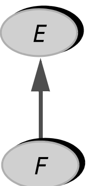


Рис. А.3. F наследует от E

Полиморфные присваивания, такие как приведенные выше, известны как «кастинг, или приведение вверх». Обратное приведение, когда объект родительского типа приводится к специфическому типу потомка, известно как «кастинг вниз» и использует синтаксис, принятый в C++:

```
f1 = (F) e1;
```

Если e1 присоединен к объекту типа F, эта операция будет присоединять f1 к этому объекту; если же нет, то кастинг станет причиной исключения в соответствии с принципом кастинга. Можно предусмотреть перехват исключения в **try**-блоке и его обработку в **catch**-блоке, но лучше избежать этого через встроенную операцию instanceof:

```
if (e1 instanceof F )
    {f1 = (F)e1;}
else
{… Обработка случая, когда e1 не обозначает объект F …}
```

Достигается эффект, подобный тесту объекта.

Универсальность

Концепции универсальности Java должны быть вам знакомы, поскольку о них шла речь при обсуждении неограниченной универсальности.

[1] но не удаляет ее.

Родовые параметры заключаются в угловые скобки <...>. Объявим:

```
public class N <G, H> {
    … Тело класса …
}
```

Класс N имеет два родовых параметра. Для построения родового порождения также используются угловые скобки:

```
N<T, U>
```

Подобно классам, интерфейсы могут быть универсальными. Ближайшим эквивалентом ограниченной универсальности является возможность объявлять формальный родовой параметр в следующей форме:

```
<? extends V>
```

Это означает, что соответствующий фактический родовой параметр должен быть потомком V.

Важным расширением механизма универсальности является возможность (отсутствующая в Eiffel) объявить универсальным отдельный метод, а не весь класс. Например, можно объявить:

```
public <G> List <G> repeated (int n, G val) {…}
```

У метода repeated два аргумента – целое n и значение произвольного типа G. В качестве результата метод возвращает список значений типа G (например, список из n одинаковых значений, заданных вторым аргументом). Рассмотрим объявление:

```
<String> repeated (27, "ABC")
```

Результатом будет список из 27 строк, с одинаковым значением «ABC».

Несколько ограничений влияют на универсальность.

- Нельзя использовать примитивные типы, такие как boolean и int, в качестве фактических родовых параметров. Вместо этого необходимо использовать объектные двойники Boolean и Integer.
- Классы, наследуемые от Exception, не могут быть универсальными.
- Для универсальных классов не допускается статический контекст.

А.5. Другие механизмы структурирования программ

Управляющие структуры Java в основном заимствованы от C и C++.

Условный оператор и оператор выбора

Условный оператор имеет форму:

```
if(boolean_expression){
```

```
    …
}
else{
    …
}
```

Разрешается опускать скобки для then- или else-части, если они состоят из одного оператора (но лучше их оставлять для облегчения будущей модификации).

Отсутствует эквивалент elseif, так что нужно использовать вложенность. Для придания визуальной структуры используются отступы:

```
if (expression) {…}
else if (expression) {…}
else if (expression) {…}
…
else statement
```

Оператор **switch** задает структуру множественного выбора, хотя он и представляет многоцелевой **goto**, а не правильную структурированную конструкцию «один вход – один выход». Множественный выбор в Java, наследованный от C, имеет вид:

```
switch (expression) {
    case value: statement; break;
    case value: statement; break;
    …
    default: statement
}
```

В этой конструкции *expression* должно быть целочисленного типа (short, byte, int или их объектные двойники) или символьного типа (char или Character). Каждое значение *value* должно быть константным выражением, вычислимым в период компиляции, и иметь совместимый тип с выражением. Если значение выражения не соответствует ни одной из этих констант, то выполняется ветвь по умолчанию – default, если она задана, в противном случае ничего не делается (в Eiffel в подобной ситуации возникнет ошибка периода выполнения).

Каждая ветвь должна заканчиваться оператором break, как показано. В противном случае управление будет «проваливаться» в следующую ветвь. Этот оператор, организующий выход из структуры, применим и для других управляющих структур – условного оператора и оператора цикла, как вскоре будет показано.

Другой подобной конструкцией является оператор continue, который может применяться в циклах. В отличие от break, он не приводит к выходу из цикла, а осуществляет переход на новую итерацию – проверку теста условия цикла.

При обсуждении управляющих структур говорилось, что лучше держаться подальше от таких goto-подобных конструкций.

Чтобы убедиться, что break или continue передает управление в нужное место, можно использовать запись их, сопровождаемую метками перехода, где в роли меток выступают идентификаторы и, конечно же, предполагается, что мы имеем дело с размеченной структурой программы:

```
label: … Управляющая структура (if, switch или loop) …
```

В то время как непомеченные операторы break и continue организуют выход из непосредственно охватывающей структуры, помеченная форма позволяет перепрыгивать на несколько уровней. Если применять эти операторы, то лучше использовать помеченную форму для уменьшения вероятности ошибок[1].

Циклы

Java предоставляет три вида циклов:

```
while (boolean_expression) statement
do statement while (boolean_expression);
for (init_statement ; boolean_expression ; advance_statement) body_statement
```

Во всех этих вариантах *boolean_expression* служит условием продолжения. Это отличается от соглашения для формы from ... until ... loop ... end, принятой в этой книге, где until-выражение используется как условие выхода. Для преобразования условия из одной формы в другую достаточно применить операцию отрицания.

Разница между первыми двумя формами цикла состоит в точке проверки условия продолжения – в начале цикла или в конце. В первом варианте тело цикла может ни разу не выполняться, во втором гарантируется, что тело цикла будет выполнено, по крайней мере, один раз.

Цикл for – наиболее общий и наиболее часто используемый. Целью *advance_statement* является обеспечение продвижения к следующей операции (в Eiffel эта часть включается в тело цикла). Приведем пример цикла в Eiffel:

```
from i := 1 until i > n loop
        …
        i := i + 1
end
```

Его эквивалент в Java:

```
for (int i = 1; i <= n; i++)
    {…}
```

Язык также обеспечивает современную форму цикла, упрощающую итерирование по коллекциям (контейнерам):

```
for (variable: collection){
...
}
```

[1] Спорный совет. Непомеченная форма – это аналог goto вперед, допускаемый некоторыми авторами. Помеченная форма ближе к настоящему goto и может создавать большую неразбериху в понимании текста: оператор перехода на одной странице, метки – на других. Лучший совет – не использовать эти конструкции.

Соответствующий эквивалент Eiffel достигается механизмом итерирования контейнерных классов с использованием агентов.

А.6. Отсутствующие элементы

Несколько механизмов, заслуживающих доверия, не представлены непосредственно в Java. Давайте посмотрим, как достичь их эффекта.

Проектирование по контракту

Java не содержит прямой поддержки проектирования по контракту (отсутствуют предусловия, постусловия, инвариант класса и инварианты цикла). В версии Java 1.4 введен оператор утверждения **assert**, используемый в форме:

```
assert boolean_expression;
```

Этот оператор (подобный оператору **check** в Eiffel) вычисляет *boolean_expression*, он ничего не делает, если вычисленное значение – *true*, в противном случае выбрасывает исключение типа AssertionError – потомка Error, следовательно, непроверяемого. Этот оператор может быть использован для вставки утверждений в определенные точки программы для мониторинга ее поведения. Если вставить такой оператор в начало тела метода (где должно выполняться предусловие) или в конец (где должно иметь место постусловие), то появляется возможность моделирования этих механизмов. Конечно, это дает лишь небольшую часть полного механизма проектирования по контракту, в частности, применения к документированию, наследованию, понятия инварианта.

Обнаружив важность отсутствующего механизма, многие группы предложили расширение языка, добавляющее контракты. Обычно эти расширения носят экспериментальный характер и используют препроцессор (инструмент, обрабатывающий расширенную версию языка и транслирующий текст расширения в стандартный Java-текст). Таких предложений десятки, наиболее известным и широко используемым вариантом является JML – Java Modeling Language, который служил основой важной работы по верификации ПО (смотри ссылку в разделе литература).

Множественное наследование

Возможность для класса реализовать множественное наследование означает возможность комбинирования нескольких абстрактных типов. Напрямую этого сделать нельзя, но мы увидим, как можно ослабить это ограничение, в частности, используя внутренние классы.

Агенты

Java не имеет агентов или подобного механизма, такого как делегаты C#. Поскольку строго типизированная структура языка исключает использование указателей функций, как в C и C++, это является серьезным недостатком для тех приложений, где желательно рассматривать операции как объекты: программирование, управляемое событиями, некоторые численные вычисления, аналогичные вычислениям интегралов, итерирование и другие приложения, рассмотренные при обсуждении агентов.

Java предложила альтернативу таким потребностям. Мы уже видели, что появились циклы со встроенным механизмом итерирования. Для большинства других случаев рекоменду-

емым Java решением является использование внутренних классов (детализированное на примере GUI в следующем разделе).

Долгое время сообщество Java отрицало необходимость введения новых средств в язык. В 1997 году появилась так называемая «белая» статья, где с убежденностью отстаивалась эта точка зрения и в целом взгляд на проблемы проектирования языка программирования. Со всем пылом демонстрировалось, что делегаты – которые были предложены для Java, но нашли воплощение в C#, – являются избыточным механизмом. В доказательство были написаны несколько примеров, где параллельно решались задачи с использованием внутренних классов и делегатов. В конечном итоге демонстрация показала превосходство того самого механизма, который в статье пытались похоронить. Это не удивительно, если вспомнить обсуждение в этой книге, где показывалось, как отсутствие агентов усложняет программу.

Понадобилось почти полтора десятка лет, чтобы проектировщики сдались: объявлено, что Java 7 включает механизм, подобный агентам, известный как замыкания.

А.7. Специфические свойства языка

Java предлагает несколько механизмов, которые отсутствуют в базисном ОО-каркасе, представленном в этой книге.

Вложенные и анонимные классы

Класс Java может быть вложенным – объявленным в тексте другого класса:

```
public class O {
    class Inner {
        … Члены Inner …
    }
            … Другие члены O, могут использовать класс Inner …
}
```

Класс Inner является примером *вложенного,* или *внутреннего,* класса. Для класса O внутренний класс подобен его членам, хотя не является ни полем, ни методом. Методы класса O могут объявлять объекты внутреннего класса и вызывать методы Inner на соответствующих объектах.

Возможна даже более удивительная вещь: если нужен класс внутри специального контекста, то можно объявить его без имени, – такие классы называются *анонимными*, пример появится ниже.

Пока можно лишь удивляться, в чем польза таких возможностей. Но они играют свою роль, имитируя отсутствующие механизмы, перечисленные выше – множественное наследование и агенты. Давайте займемся рассмотрением двух приложений.

Предположим, что необходимо написать класс R как наследника двух классов P и Q. Язык позволяет выбрать только одного, пусть P, от которого R официально наследует. Для Q обычное решение состоит в использовании клиентского отношения. Внутренний класс предлагает некоторое улучшение этой техники. Можно добавить в R внутренний класс S, наследующий от класса Q. Это позволяет членам R использовать члены Q через нотацию, квалифицируемую именем класса, – S.some_member_of_Q:

```
public class R extends P {
```

```
    class S extends Q{               // S - внутренний класс R
       … Члены S, включая возможно переопределенные члены Q …
       }
    … Другие члены R …
    // Здесь можно использовать члены S,
    // включая наследуемые от Q в форме S.f(…)
}
```

Поскольку R предлагает функциональность Q, часто удобно добавить в R обертывающие методы, задавая для каждого **public**-метода f из Q соответствующий обертывающий метод в форме:

```
T f (T1 a1, …)// С той же сигнатурой и результатом, что у Q.f
    {S.f (a1, …)}                                              [1]
```

Преимущества и ограничения этой техники, имитирующей множественное наследование, понятны. Положительно то, то обеспечивается прямой доступ ко всем методам отвергнутого родителя (здесь Q) и позволяется переопределение его методов. Но это может приводить к дублированию кода (если используется схема [1]), и не достигается симметрия множественного наследования. Один родитель – настоящий, другой – бедный дядя, которого пустили жить на чердак. В частности, нельзя полиморфно использовать экземпляр R как экземпляр Q, только P имеет такую привилегию. Дяде не дают забыть, что ему не все позволено.

Рассмотрим теперь, как промоделировать агентов на примере GUI-программирования. Предположим, что требуется для любого появления события определенного типа, например нажатия левой кнопки мыши на некотором интерфейсном объекте, таком как кнопка *OK_button*, включить выполнение некоторой программы *perform* вашей системы, принимающей в качестве аргументов два целых числа, представляющих координаты мыши. Мы видели, как это делается с использованием агентов:

```
OK_button.left_click.subscribe (agent perform)                  [2]
```

Для понимания этого обсуждения нужно быть знакомым с концепциями, которые были описаны при рассмотрении программирования, управляемого событиями (глава 18).

Давайте посмотрим, как достичь того же эффекта **[2]** в Java, используя общий подход библиотек Java GUI, таких как Swing. Мы не будем использовать их точную терминологию (цель не в том, чтобы научить пользоваться конкретной библиотекой – для этого можно найти много Web-страниц – но показать, что находится под капотом и как мотор фактически работает). Прежде всего, необходим интерфейс, связанный с типом события, например:

```
interface ClickListener {
    void process(ClickEvent e);
}
```

На каждый тип события строится один такой интерфейс. У него только один метод, названный process, который обозначает операцию, выполняемую в качестве реакции на появление события данного типа. Аргумент e представляет событие. При возникновении события в соответствующем модуле создается объект подходящего типа (здесь ClickEvent), содер-

жащий аргументы события, такие как координаты мыши. Мы предполагаем, что аргументы доступны для события e, как e.args.

Как свидетельствует имя интерфейса, его экземпляры (точнее, экземпляры его потомков) представляют объекты, подписанные на соответствующий тип события. «Слушатель» (listener) – это еще один термин для обозначения «наблюдателей», или «подписчиков», события.

Внутренне общий механизм, позволяющий уведомлять подписчиков о событиях, такой же, как в образце «Наблюдатель» и, в более простой форме, библиотеке «Event». Для каждого типа события хранится список подписчиков; когда событие возникает, выполняется обход списка, так, чтобы каждый мог выполнить предписанную операцию. Все, что нам нужно посмотреть, это как такой класс, скажем, U, подписывается на тип события, задавая собственную операцию – реакцию на событие. Здесь можно использовать анонимный вложенный класс:

```
class U {                                                              [3]
    Button b;
    build () {
        okButton=new Button(…);
        okButton.addListener(
            new ClickListener(){
                public void process(ClickEvent e){
                            // Код, который должен выполняться для e, например:
                        perform(e.args);
                }
            }
    }
        … Другие члены U …
}
```

Базисная схема добавления слушателя – такая же, как и в предыдущих главах: добавляется элемент в список слушателей, здесь это делается в форме okButton.addListener(obs). В образце наблюдатель obs является экземпляром класса наблюдателей, и необходимо было определять такой класс для каждого возможного типа события и наблюдателя. Механизм агентов существенно упрощает ситуацию, позволяя просто записывать obs как **agent** *perform*, где *perform* инкапсулирует метод, который требуется вызвать, вместе с любыми ассоциированными объектами. Эта схема близка к механизму «Наблюдатель», но с важным улучшением: нет необходимости загромождать нашу систему новыми классами, играющими только локальную роль. Вместо этого можно использовать анонимные вложенные классы.

Так как ClickListener – это интерфейс, то к нему неприменима конструкция new для создания объекта. Необходимо определить класс, реализующий интерфейс ClickListener, и сделать obs экземпляром этого класса. Так как такой класс нужен только в фиксированном контексте, можно определить его прямо на месте. Как это делается, показано в программе **[3].** Так как класс больше нигде не будет использоваться, он не нуждается в имени.

Правильные (эффективные в терминах этой книги) потомки интерфейса ClickListener должны реализовать единственный метод этого интерфейса – process. Наша реализация вызывает желаемый метод нашей системы – perform, передавая ему аргументы события.

Преимущество в том, что можно использовать класс непосредственно в желаемой области, не засоряя глобальное пространство имен. Кроме того, внутренние классы имеют доступ

ко всем членам охватывающего класса, включая закрытые, private-члены. Это может быть полезным, например, если U является частью GUI-приложения, и коду подписчика – здесь *process* необходимо изменить элементы интерфейса пользователя.

Эта техника, однако, очевидно имеет одно из ограничений, которое раньше называлось синдромом «много маленьких оберток», заставляющих обертывать операции в небольшие классы. Здесь нет необходимости придавать таким классам статус первого ранга, они остаются локальными и анонимными, но все же их нужно определять – один интерфейс на каждый тип события.

Нет необходимости в подробных обсуждениях. Достаточно беглого взгляда на схемы **[2]** и **[3]** для понимания разницы в выразительной силе, но, как говорилось, кажется, что нужное сообщение дошло и до Java-сообщества.

Преобразование типов

Java поддерживает преобразования значений между различными примитивными типами.

Там, где нет потери точности в преобразованиях – byte в short, short в int, int в long, char в int, int в double, float в double – можно использовать прямое присваивание, например l = s; где s типа short, а l типа long.

Эта возможность также применима (как в большинстве языков программирования) к некоторым случаям, в которых потеря точности считается приемлемой: int в float, long в float, long в double.

Для случаев с более важной потерей точности, таких как преобразование значений с плавающей точкой в целочисленные (требующие округления или другой аппроксимации), необходимо явное задание преобразования типов, чтобы показать, что вы отвечаете за то, что делаете. Вот пример:

```
float s;                    // Вещественные данные с однократной точностью
double d = 9.9;             // С двойной точностью
s = (float) d;
```

Как мы видели, существуют взаимные преобразования между примитивными типами и их объектными двойниками. В отличие от Eiffel, в Java нет возможности определения преобразования между произвольными типами[1].

Тип перечисление

Тип, задающий перечисление, дает возможность определить переменные, с возможными значениями из конечного множества предопределенных значений, как в следующем примере:

```
enum CardColors {Spades, Hearts, Diamonds, Clubs}
```

Ссылаться на значения можно, используя нотацию с точкой, используя имя класса в качестве префикса – CardColors.Spades.

Внутренне тип, такой как CardColors, наследует от библиотечного класса Enum. Так как это класс, можно добавлять конструкторы, атрибуты и методы.

[1] Конечно, всегда можно самому программисту задать преобразование между любыми типами, если он придумал алгоритм преобразования. Конечно же, в Java определены взаимные преобразования типа String и примитивных типов, необходимые при вводе-выводе данных.

Методы с переменным числом аргументов – Varargs

Начиная с версии 5.0, в Java можно задавать методы с переменным числом аргументов или «varargs» (не используя для этой цели массив или другую коллекцию – стандартный прием в отсутствие специфического механизма языка). Соглашение простое – после типа последнего аргумента можно задать многоточие, означающее, что этому аргументу может соответствовать произвольное число фактических аргументов:

```
public void m(T1 a1, T2 a2, String… s)
```

Интерпретация записи такова: при вызове метода после двух фактических аргументов могут идти ноль, один, два и более аргумента типа String. Метод может иметь не более одного такого varargs-аргумента, и он должен идти последним в списке формальных аргументов.

Эффект такой же, как если бы последний аргумент был массивом (здесь массив String). Метод мог бы тогда использовать свойства массива s.length, чтобы найти, сколько значений было фактически передано, и s[i] для доступа к ним[1].

Аннотации

В версии 5.0 в язык введены аннотации, представляющие механизм добавления структурированной информации – аналог конструкции **note** в Eiffel и атрибутов в C#.

Информация, содержащаяся в аннотации, не влияет на семантику самой программы, но она может представлять интерес для других средств, например, для инструментария, управляющего проектом.

Аннотации основаны на интерфейсах и объявляются с ключевым словом @interface. Например, в организации может быть предусмотрен стандартный способ поставки класса, включающий информацию об авторе, дате модификации, номере версии. В этом случае можно объявить аннотационный интерфейс:

```
public @interface ChangeInfo {
    string author;
    string last;
    string revision;
}
```

Тогда можно поставить класс или метод, содержащий эту информацию:

```
@ChangeInfo{
    author="Caroline",
    last="24 December 2009",
    revision="6.7"
} public void r {…}
```

Библиотеки Java позволяют выполнять процесс «отражения», обеспечивая программу информацией о ее собственной структуре. Используя отражение, можно получить доступ к аннотациям, связанным с программным элементом. Вот как это делается:

---

[1] Эффект все-таки разный. Если формальный аргумент – массив, то и фактический должен быть массивом. А для varargs-аргумента можно задать последовательность с произвольным числом элементов. Эта возможность активно используется, например, при выводе на печать нескольких переменных.

```
x.getAnnotations()
```

Здесь x (полученный через отражение) представляет класс или метод.

А.8. Лексические и синтаксические аспекты

Символы Java используют кодировку Unicode. Подобно Eiffel и большинству других современных языков, Java применяет «свободный формат»: разделительные символы (пробелы, табуляция, переход на новую строку) эквивалентны и служат только для разделения лексем.

В отличие от Eiffel, идентификаторы чувствительны к регистру. Они могут быть произвольной длины, но не могут начинаться с цифры, включать такие символы, как / или - (слеш и дефис). При написании многословных идентификаторов в Java обычно используется «Кэмел»-стиль: aCamelCaseName.

Комментарии имеют две формы. В первой из них комментарий начинается с двух слешей – //, и распространяется до конца строки. Он многократно использован в примерах. В другой форме комментарий распространяется на несколько строк, обрамленных парами символов /* и */. Возможны также документируемые комментарии, позволяющие автоматически создавать документацию инструментом Javadoc. Такие комментарии начинаются тройкой символов /**

Отметим, на случай, если вы этого не заметили (тогда следует перечитать это приложение сначала), что язык Java сохранил два синтаксических соглашения C and C++: заканчивать каждое объявление и оператор символом точки с запятой, использовать знак равенства в присваиваниях и двойное равенство для операции эквивалентности.

Ключевые слова

Следующие имена резервированы в языке Java:

```
abstract, boolean, break, byte, case, catch, char, class, const, continue, default,
do, double, else, extends, final, finally, float, for, goto, if, implements,
import, instanceof, int, interface, long, native, new, null, package, private,
protected, public, return, short, super, switch, synchronized, this, throw, throws,
transient, true, try, void, volatile, while
```

Имена const и goto появляются в этом списке, хотя и не используются в настоящее время.

Операции

Вот таблица операций с их приоритетами:

Доступ, вызов	[] ()
Другие унарные	+ - ~! new ()
Арифметические	* / %
Сдвиг	<< >> >>>
Эквивалентность	== !=
Тернарная	cond ? expr1:expr2
Присваивание	^= |= <<= >>= >>>=
Постфиксные	expr++ expr—
Префиксные	++expr —expr

Аддитивные + -
Отношения < > <= >= instanceof
Логические & ^ | && ||
Присваивание = += –= *= /= %= &=

A.9. Библиография

James Gosling, Bill Joy, Guy Steele and Gilad Bracha: The Java Language Specification, third edition, Addison Wesley, 2005.

Как это часто бывает, описание языка от его создателей превосходит позднейшие производные работы. Должно быть прочитано каждым, кто интересуется языком Java.

Joshua Block: Effective Java, second edition, Prentice Hall, 2008.

Bruce Heckle: Thinking in Java, fourth edition, Prentice Hall, 2006.

Cay S. Hearthstone and Gary Corneal, Core Java, Volume 1 (Fundamentals), eighth edition, Prentice Hall, 2007.

Три широко используемых учебника.

java.sun.com/reference/docs

Документация онлайн от корпорации Sun.

www.eecs.ucf.edu/~leavens/JML/

Это домашняя страница JML, на которой можно найти ссылки на многочисленные работы по расширению Java проектированием по контракту.

B

Введение в C# (по материалам Бенджамина Моранди)

При появлении языка C# в 1999 Microsoft представлял язык следующим образом.

Идеальное решение для C и C++ программистов, обеспечивающее быструю разработку в сочетании с мощью доступа ко всей функциональности платформы. Программисты получают окружение, полностью синхронизированное с появляющимися Web-стандартами и обеспечивающее простую интеграцию с существующими приложениями. В дополнение появляется возможность при необходимости создавать код на низком уровне.

Язык C# – это современный ОО-язык, позволяющий программистам быстро строить широкий круг приложений для новой .NET платформы, предоставляющей полный набор инструментария и сервисов, которые необходимы как для вычислений, так и для коммуникаций.

Элегантно спроектированный ОО-язык C# является великолепным выбором при построении архитектуры компонентов – начиная от высокоуровневых бизнес-объектов до приложений системного уровня. Используя простые конструкции языка C#, эти компоненты могут быть конвертированы в XML Web-сервисы, вызваны затем в Интернете из любого языка, выполняемого на любой операционной системе.

Рис. B.1. Андерс Хейлсберг (разработчик C#), 2007

Большинство читателей этой книги предпочитают нормальный русский язык – поэтому стоит дать перевод этого пышного представления: «C# – это Java плюс делегаты (объекты в духе агентов) и несколько низкоуровневых механизмов, заимствованных от C++». C# был ответом Microsoft в конкурентной борьбе с компаниями, поддерживающими Java, в частности Sun Microsystems и IBM. Язык чрезвычайно близок к Java.

Эта характеристика остается во многом справедливой и сегодня, хотя C# эволюционировал своим собственным путем и ввел несколько интересных инноваций, не имеющих анало-

гов в Java. На момент написания этого текста C# (версия 3.0) является мощным языком, и здесь мы рассмотрим только его основы .

Для изучения C# знание Java полезно, но не требуется. Это приложение не предполагает, что вы прочли описание Java, приведенное в предыдущем приложении (как следствие, повторяются некоторые рассуждения, когда рассматриваются разделяемые концепции). Подобно другим приложениям, язык не рассматривается с чистого листа, – обсуждение предполагает знакомство с программистскими концепциями, введенными в этой книге. Описание языка сопровождается сравнением с соответствующими механизмами Eiffel.

В.1. Окружение языка и стиль

C# (произносится «С шарп») тесно связан с окружением Microsoft .NET, платформой для разработки и выполнения ПО, использующей виртуальную машину. В предыдущих обсуждениях отмечалась роль виртуальных машин и их преимущества для реализации языков высокого уровня.

.NET, CLI и взаимодействие с языком

В то время как виртуальная машина Java – JVM – была спроектирована специально для поддержки этого языка (хотя позднее она использовалась для реализации других языков программирования), главная цель проекта платформы .NET состояла с самого начала в поддержке нескольких языков. Это решение отражается как в имени виртуальной машины – Common Language Runtime (CLR) – «Общеязыковая Среда Выполнения», так и в поддержке взаимодействия API, Common Language Infrastructure (CLI) – «Общеязыковой Инфраструктуры», которая теперь является международным стандартом.

Частично причина была в том, что Microsoft еще до .NET обеспечил реализацию нескольких языков, прежде всего, Visual Basic (VB), C++ и JScript для клиентских Web-приложений. VB популярен на массовом рынке и часто используется для разработки приложений по настройке офисных документов. Компания не могла, естественно, предложить соответствующим программистским сообществам бросить свои любимые языки и перейти на новый бренд. Она смогла обеспечить общую базу для взаимодействия и будущей эволюции. .NET и CLR/CLI были способны с самого начала обеспечить реализацию четырех поддерживаемых Microsoft языков (помимо трех упомянутых еще и C#), а также языки, разрабатываемые другими компаниями, включая Eiffel (с самого начала введения .NET в 1999) и Cobol, язык прошлых лет, но все еще важный для многих бизнес-приложений.

Языковая открытость среды означала не только доступность нескольких компиляторов, но и влекла высокую степень взаимодействия между программами, написанными на разных языках. В этом роль Общеязыковой инфраструктуры: обеспечить стандартное множество механизмов, которые могут быть использованы в любом языке. CLI представляет объектную модель, близкую ОО-языку, но без синтаксиса. Эта модель задает множество хорошо определенных механизмов – это ОО механизмы, изучаемые в этой книге: классы, их компоненты, наследование, универсальность, система типов, объекты, политика динамического создания объектов и сборка мусора – для которых CLI-проект предложил несколько специфических проектных решений.

При условии, что .NET-языки не слишком отклоняются от этих решений, они могут достичь степени взаимодействия, неслыханной в дни, предшествующие .NET. В частности, классы, написанные на разных языках, могут взаимодействовать друг с другом, используя как отношения наследования, так и клиентские. Например, Eiffel-класс может наследовать от класса C#, возможно и обратное наследование. Это просто предполагает, что компилято-

ры следуют единым CLI-правилам для поставщиков и клиентов сборок (целевых модулей, создаваемых .NET-компиляторами).

Эта схема взаимодействия доказала свою успешность (несмотря на существующую тенденцию производителей ПО игнорировать правила CLI-совместимости). Она позволяет каждому языку сохранять свою индивидуальность, пока ее можно отобразить в объектную модель CLI. Например, реализация Eiffel должна моделировать множественное наследование – не поддерживаемое напрямую CLI – через специальное использование CLI-механизмов (множественное наследование интерфейсов, концепцию, обсуждаемую позже в этом приложении).

Любимый сын

В сообществе языков, как и у людей, все языки равны, но некоторые равны более других. Язык C# – это любимый сын: его объектная модель наиболее тесно связана с CLI. VB .NET, который схож с предыдущей версией только синтаксически, является еще одним претендентом на звание «любимого». CLI-совместимая версия C++ – «управляемый C++» – существенно отличается от обычного C++. Ограничения необходимы, чтобы язык мог принимать участие в играх .NET-взаимодействия. Фактически, семантика C# определялась семантикой CLI, хотя последующие версии ее существенно расширили. Синтаксис языка соответствует традиции C, C++, Java, включая завершение операторов символом точки с запятой и применением фигурных скобок для окаймления блоков программы.

B.2. Общая структура программы

Базисными элементами C# программы являются классы и структуры, организованные в виде нескольких программных файлов.

Классы и структуры

C#-классы (ключевое слово *class*) и структуры (ключевое слово *struct*) задают описание множества возможных объектов периода выполнения. Объекты обладают свойствами, и к ним применимы методы. Общая форма объявления такова:

```
class name {
    … Объявление компонентов …
}
```

При объявлении структуры вместо ключевого слова class используется слово struct.

Компоненты могут быть разные. Это может быть:

- поле, соответствующее атрибуту Eiffel;
- константа, частный случай поля;
- метод, реализованный в виде процедуры или функции;
- метод-свойство, поле, сопровождаемое возможными методами – геттером и сеттером;
- операция, функция с синтаксисом операции;
- конструктор, процедура создания – метод, применяемый для создания объектов;
- деструктор, редко применяемый метод для освобождения ресурсов, возможный только для классов;
- событие, которое связано с делегатами и программированием, управляемым событиями;
- индексатор;
- вложенный тип.

Структура является упрощенной формой класса без возможности наследования. Остальная часть обсуждения фокусируется на классах, но большинство свойств, не связанных с наследованием, применимо и к структурам[1].

Классы и структуры группируются в сборки (понятие, соответствующее кластеру в Eiffel)[2].

Выполнение программы

Каждая выполняемая программа должна иметь по меньшей мере один метод, называемый Main и помеченный как статический (static – понятие, поясняемое в следующем разделе). Выполнение программы начинается с выполнения этого метода. Можно написать классический пример «Hello world» с одним классом и Main-методом:

```
public class Program {
    static void Main(string[] arguments) {
       System.Console.WriteLine("Hello world!");
    }
}
```

Main может не иметь аргументов или, если выполнение нуждается в аргументах, предоставляемых пользователем, иметь один аргумент, представленный массивом строк (string[]). Метод может не возвращать результат, или возвращать целочисленное значение, которое обычно рассматривается как статус, сигнализирующий об уровне ошибок, если метод завершается с ошибками.

В.3. Базисная ОО-модель

Многие концепции C# совпадают с теми, что мы видели при изучении этой книги. Но есть некоторые вариации.

Статические компоненты и классы

Одна из концепций C#, уклоняющаяся от строгого ОО-стиля, который используется в этой книге, состоит в поддержке статических компонентов класса и классов в целом.

Обычно для использования компонента класса необходим целевой объект. Стандартная ОО-нотация доступа к компоненту имеет вид target.member (возможно, с передачей аргументов компоненту), где target обозначает целевой объект. Текущий объект (**Current** в Eiffel), в C# имеет имя **this**, которое, как и в Eiffel, можно опускать, когда из контекста ясно, что речь идет о текущем объекте.

В C# разрешается объявлять статические члены, не требующие объекта и вызываемые как C.member, где C – имя класса. Определение статического компонента, например статического метода, может использовать только статические компоненты.

Класс в целом может также быть объявленным как статический, если все его компоненты статические. Тогда невозможно создать экземпляры этого класса. Статический класс мо-

[1] Главное отличие классов от структур состоит в том, что класс определяет ссылочный тип, а структура – развернутый. Значения ссылочного типа разделяют память – на один и тот же объект может указывать несколько ссылок, имена ссылок являются синонимами. Объект развернутого типа ни с кем свою память не разделяет. Все примитивные типы – арифметический и другие – реализованы как структуры.

[2] Для программиста классы, структуры и другие частные случаи – интерфейсы, делегаты, перечисления – группируются в проекты, преобразуемые в сборки в результате компиляции проекта.

жет быть удобен, например, для группирования множества объектно-независимых общих свойств, таких как математические функции.

Уже упоминалось, что метод Main должен быть статическим; причина в том, что на старте выполнения не существует объекта, который мог бы вызвать метод. В Eiffel проблема решается за счет того, что выполнение определяется как создание «корневого объекта», к которому применяется «корневая процедура»[1].

Статус экспорта

Для скрытия информации каждый тип и компонент имеет уровень доступности, определяя права клиентов на доступ. Цель та же, что и в Eiifel: механизм скрытия информации, включая селективный экспорт, но с грубой гранулярностью, поскольку в C# нельзя создать список ВИП-персон – классов, которым будет доступен некий компонент класса. Тремя возможными квалификаторами являются:

- public: доступен в любом коде;
- internal: доступен в коде той же сборки;
- **private**: доступен коду самого класса или структуры. Это статус по умолчанию для компонентов класса[2].

Применимы некоторые ограничения: класс может быть только internal (по умолчанию) или public, если он не является внутренним классом (классом, объявленным внутри другого класса), который может быть также и private. Деструкторы не могут иметь модификаторов доступа. Операции, определенные программистом, должны быть static и public. Доступность компонентов не может превосходить доступность класса. Не трудно видеть смысл, стоящий за каждым из этих правил[3].

Поля

C#-поля соответствуют атрибутам. В этой книге (вне приложения) используется другая терминология: под полем (динамическим понятием) понимается составляющая объекта, соответствующая компоненту генерирующего класса – атрибуту (статическое понятие). В C# один термин применяется для обоих понятий.

При объявлении поля задается его тип (перед именем поля, как в T f, вместо *f*: *T* в Eiffel). Объявление может включать инициализацию поля, используя для этого символ присваивания =, после которого может идти константное выражение, вычислимое в момент компиляции и не содержащее других полей, отличных от констант и статических полей. Вот пример объявления двух полей:

```
class A {
    public string s1 = "ABC";
    public readonly string s2 = "DEF";
    … Other member declarations …
}
```

[1] По поводу статических компонентов класса в Java и C# смотри мой комментарий в приложении по Java. Статический конструктор C# создает статический объект, который и является целью вызова статических компонентов. В статический конструктор можно добавить свой код, например, для определения специфических констант класса. Выполнение программы C#, как и в Eiffel, можно рассматривать как создание статическим конструктором корневого статического объекта, который и вызывает корневую процедуру – Main.

[2] Есть, конечно, и четвертый квалификатор – protected, – позволяющий получить доступ потомкам класса. Возможна и комбинация protected, internal. Позже об этом будет сказано.

[3] Важным ограничением для C# является то, что компоненты интерфейсов объявляются без указания квалификаторов доступа.

Заметьте: точка с запятой завершает все объявления и операторы. Квалификатор readonly защищает поля от присваивания, за исключением инициализации в момент объявления, как здесь, или в конструкторе.

В отличие от Eiffel, экспорт полей без статуса readonly дает клиентам право на чтение и запись. Для приложений, признающих преимущества скрытия информации, это означает, что поля должны иметь статус по умолчанию private и при необходимости снабжаться специальными методами доступа – геттером и сеттером. C# упрощает их написание, введя понятия метода – свойства, изучаемого ниже.

Базисные типы

C# обеспечивает несколько встроенных типов:

- bool, представляющий булевские значения;
- char, представляющий 16-бит Unicode-символы. Константа char записывается в одиночных кавычках, как 'A';
- string, представляющий последовательность из нуля или более символов char. Строковая константа записывается в двойных кавычках, как «ABC»;
- целочисленные типы: sbyte (знаковый 8-бит), byte (беззнаковый 8-бит), short (знаковый 16-бит), ushort (беззнаковый 16-бит), int (знаковый 32-бит), uint (беззнаковый 32-бит), long (знаковый 64-бит), ulong (беззнаковый 64-бит);
- вещественные (с плавающей точкой) типы: float, double и decimal, представляющие 32-бит, 64-бит и 128-бит IEEE-числа с плавающей точкой.

Тип object является предком всех типов (аналог *ANY* в Eiffel).

Ссылка null (void) записывается как null.

Ссылки и значения

Каждый C# тип является ссылочным или значимым типом. Отличия те же, что и для Eiffel. Переменная значимого типа непосредственно обозначает значение, которое может быть простым значением только что рассмотренного типа (встроенные типы, за исключением string и object, являются значимыми типами) или сложным объектом. Переменная ссылочного типа обозначает ссылку на объект.

Перейти от значения к ссылке можно, используя операцию, называемую боксингом, или упаковкой:

```
int i; object o;
i = 1;
o = i;//Boxing: Создает объект, обертывающий значение, присоединяет o к нему
```

Как показывает этот пример, операция боксинга выполняется автоматически при присваивании ссылке переменной значимого типа. Обратная операция – распаковка – должна выполняться явно, с использованием кастинга – приведения к типу:

```
i=(int)o;//Распаковка: Получение целого, хранимого в o, и присваивании его i.
```

Константы

Поле может быть объявлено константой, как const, указывающее, что для всех экземпляров класса оно сохраняет значение, заданное при инициализации. Значение может быть лите-

ральной константой (манифестным целым, строкой и так далее) или константным выражением, включающим ранее определенные константы:

```
public const string s3 = "ABC- ";
public const string s4 = s3 + "DEF";   // Значение: "ABC-DEF"
```

Так как константы относятся к статическим объектам, для доступа к ним используется имя класса A.s4, если приведенное объявление константы появилось в классе A.

Заметьте разницу между const- и readonly-полями. Значения первых должны быть заданы при объявлении, значения вторых – могут быть заданы в конструкторе (например, в статическом конструкторе).

Методы

Метод в C# может быть реализован процедурой (возвращающей тип void – результат отсутствует) или функцией, возвращающей результат, тип которого отличен от void. Вот примеры, иллюстрирующие некоторые важные возможности:

```
class B {
    public void p(int arg1, ref int arg2) {… arg2 = 0;}    // Процедура
    public string f() {… return "ABC";}                     // Функция
    public static string sf() {… return "DEF";}             // Статическая функция
}
```

По умолчанию аргументы передаются «по значению» (как в Eiffel), в этом случае формальный аргумент представляет копию фактического аргумента (в зависимости от типа – ссылочного или значимого – копия может быть ссылкой или полным объектом). Аргументы можно передавать «по ссылке», снабдив их описателем ref. В этом случае присваивание аргументу, такому как arg2, в процедуре p, будет модифицировать и фактический аргумент.

Фактический аргумент, соответствующий ref-формальному аргументу, должен также специфицироваться как ref при вызове:

```
B v = new B();
int x = 1;
int y = 1;
v.p(x, ref y);         //Не изменяет x, но значение y становится равным нулю
```

Объявление локальных переменных может появляться в теле метода, предваряя их использование. Имена не должны совпадать с именами формальных аргументов и других локальных переменных.

Имена локальных переменных и формальных аргументов могут совпадать с именами полей класса, имея приоритет. Конфликт не возникает, поскольку поле класса можно квалифицировать именем текущего объекта this, как в примере:

```
int a;                              // a - имя поля класса
r (int a) {this.a = a;}       // и формального аргумента
```

Метод может получить доступ к полю, используя нотацию this.a. Для конструкторов C# типичной практикой является именовать аргумент, служащий для инициализации поля, именем этого поля. Лучше избегать этого и выбирать разные имена для каждой цели.

Перегрузка

C# допускает перегрузку методов: несколько методов класса могут иметь одинаковые имена, если их сигнатуры различны (отличаются числом аргументов или их типами, включая и описатель ref, входящий в сигнатуру). Тип результата в сигнатуру не входит.

> Предыдущий методологический комментарий применим и здесь. Имена не являются дефицитным ресурсом. Перегрузка, однако, является распространенной практикой в C# и требуется для конструкторов, что будет обсуждаться ниже.

Методы-свойства

Политика экспорта, как отмечалось, не различает доступ на чтение и на запись. Это значит, что поле никогда не следует экспортировать, так как это позволило бы клиентам выполнять прямые присваивания $x.a = v$ полю с именем a, нарушая все принципы скрытия информации. ОО-решение в этом случае обеспечивает сеттер-процедура и геттер-функция (в которой нет необходимости в Eiffel, так как при экспорте гарантируется статус «только для чтения»). В C# написание геттеров и сеттеров стандартизовано благодаря введению понятия метода-свойства. Вот образец использования метода-свойства для закрытого поля:

```
class C {
    private string a;            // Закрытое поле
    public string ap {           // Метод-свойство
       get {return a;}           // Геттер
       set {                     // Сеттер
          a = value;             // Изменение значения поля
          … Возможно, другие операторы …
}}}
```

Этот механизм использует три ключевых слова get, set и value. Объявляются два специальных метода с именами get и set для доступа к атрибуту, в случае сеттера – через синтаксис присваивания:

```
C x = new C();
string b;
b = x.ap;                // Использование геттера
x.ap = "ABC";            // Использование сеттера
```

> Эффект подобен тому, что достигается в Eiffel спецификацией присваивания. Механизм Eiffel не требует геттера, который на практике сводится к чтению поля. Сеттер, чаще всего, – нечто большее, чем присваивание, (например, при изменениях поля может требоваться запись в журнал регистрации), так что нормально записывать его явно как процедуру.

Конструкторы

Процедуры создания, используемые для создания и инициализации объектов, называются в C# конструкторами. Конструктор может быть:

- конструктором экземпляра, динамически создающим и инициализирующим объект;
- статическим конструктором, создающим и инициализирующим статический объект.

Следующий класс содержит пример каждого типа:

```
class D {
    public D(string a) {                          // Конструктор 1: экземпляра
       … Инициализирует поле, обычно использует аргумент a …
    }
    static D() {                                  // Конструктор 2: статический
       … Инициализирует статические поля …
}
}
```

Конструкторы не имеют собственных имен, используя имя класса, основываясь на перегрузке и разрешая конфликты через отличия в сигнатурах, если есть более одного конструктора.

> Иногда возникают проблемы, например, классу POINT, описывающему точку на плоскости, полезно иметь два конструктора make_cartesian и make_polar, задающих координаты точки в декартовой и полярной системе координат. Оба конструктора имеют одинаковую сигнатуру – два аргумента типа float. Для разрешения конфликта одному из конструкторов приходится добавлять фиктивный аргумент.

Объявление конструктора не специфицирует возвращаемый тип (void тоже не задается). У статического конструктора не может быть никаких других модификаторов.

Создание нового объекта основано на операции new (**create** в Eiffel) и вызове конструктора экземпляра. Вот пример:

```
D x = new D("ABC");
```

В C# нет различия между объявлениями (статикой) и операторами (динамикой), позволяя в объявлениях создавать и инициализировать объекты, выполняя операцию new с вызовом конструктора, создающего объект – экземпляр класса D в примере.

Это, однако, далеко не полная история о конструкторах экземпляра и создании экземпляра. Детальная спецификация (дается ниже при обсуждении наследования) объясняет, что ваш вызов конструктора может в результате приводить к вызову других конструкторов, создавая цепочку вызовов, которая должна включать вызов конструктора каждого предка класса.

Статический конструктор, существующий в единственном экземпляре, без аргументов, что отражено в примере, выполняется до того, как потребуется экземпляр класса или доступ к статическому элементу класса. Это позволяет инициализировать свойства, связанные с классом в целом, прежде чем появятся специфические экземпляры (как это делается в Eiffel через однократные once-функции). Представьте экземпляр системы, фиксирующей ошибки, где ошибки записываются в специальный журнал (файл). Первое появление ошибки приводит к созданию и открытию этого файла.

Деструкторы

В C# предполагается существование сборщика мусора: в то время как создание объектов выполняется явно при задании операции new, освобождение памяти для неиспользуемых более объектов возлагается на автоматический механизм – сборщик мусора (*Garbage Collector – GC*).

Иногда можно попросить *GC* выполнить специфическую операцию, помимо освобождения памяти. Типичным примером является работа с файлами. Всякий раз, когда закончена

работа с объектом, представляющим файл, требуется закрыть физический файл, связанный с объектом (в Eiffel можно для этих целей вызывать процедуру *dispose*, которую *GC* будет выполнять при освобождении объекта).

Деструкторы C# отвечают этим потребностям. Имя деструктора ~C, где C – имя класса. Здесь нет перегрузки, так как деструктор существует, если он задан, в единственном экземпляре. Он не имеет аргументов, не возвращает значения, не имеет модификаторов, таких как static:

```
class File {
    … Другие компоненты, включая конструкторы …
    ~File() {
        … Операторы, закрывающие, например, физический файл …
}
}
```

Операции

Операция, определяемая программистом, – это статический метод, где знак операции выступает в качестве имени метода, как в примере:

```
class E {
    public static E operator +(E a, E b) {
        … Вычисление результата exp, возвращаемого в операторе return …
        return exp;
        }
}
```

Операция может быть вызвана в инфиксном синтаксисе, свойственном операциям в математике, как x + y, где *x* и y типа E (это аналогично alias-компоненту Eiffel). Операции встроены в базисные типы, такие как int и float. Имена (знаки) операций и их приоритеты фиксированы (в отличие от Eiffel, нельзя определять собственные знаки операций). Главными доступными знаками операций являются:

```
+       -       !       ~       ++      --
```

Здесь «!» – отрицание для булевских, «~» – отрицание для целых (взаимное обращение 0 и 1 в бинарном представлении числа). Операции «++» и «– –» имеют побочный эффект: x++ возвращает значение x, затем увеличивает x на 1; ++x также увеличивает x, возвращая увеличенное значение; (аналогичная семантика у операций x– – и – –x). Как вы знаете, операции с побочным эффектом – не очень хорошая идея, используйте их на собственный страх и риск.

Бинарными операциями являются:

```
+   -   _   /   %   ^           // Арифметика (% - остаток от деления нацело)
<   >   <= >=                   // Операции отношения  (дают булевский результат)
== !=                           // Эквивалентность (равно и не равно)
&   ^   |                       // Булевские строгие (and, xor, or)
<<   >>                         // Побитовый сдвиг (влево, вправо)
&&   ||                         // Булевские полустрогие (and, or)
```

Как и в других языках, наследуемых от С, знак равенства означает присваивание, а эквивалентность задается двумя знаками равенства. Некоторые из приведенных выше операций применяются к побитовому представлению целых. Строгие булевские операции применимы не только к булевским значениям, но и к целым, применяя операцию к каждой паре соответствующих битов. Операции сдвига сдвигают битовое представление влево и вправо на число позиций, задаваемое вторым операндом, при этом биты, выходящие за края сетки, отведенной целому, исчезают, а свободные места заполняются нулями. Сдвиг влево m << n эквивалентен умножению m на 2^n, сдвиг вправо – делению.

Можно использовать перечисленные знаки для определения операций в собственных классах, исключением являются знаки полустрогих операций. Механизм перегрузки используется и для операций. В случае операций сравнения перегрузка должна идти парами – если перегружается операция «меньше» (<), то необходимо определить и «больше» (>). Аналогично и для других операций сравнения.

Кроме того, C# поддерживает следующие неперегружаемые операции:

```
[…]                                              // Получение элемента массива
(…)                                              // Кастинг
+=      -=      _=      /=      %=      ^=       // Присваивание
&=      |=      <<=        >>=                   // Присваивание
```

Присваивание x += 1 это сокращенная форма записи x = x + 1. Аналогичный смысл и у других операций присваивания с операциями.

Массивы и индексаторы

Для объявления массива с одной или более размерностью используется нотация с квадратными скобками:

```
string[] a;                                      // Одномерный массив
string[,] b;                                     // Двумерный массив
```

Для задания многомерных массивов используются запятые, как показано в примере. Число запятых, увеличенное на 1, определяет размерность массива. Нижняя граница индекса по каждому измерению фиксирована и равна 0. Поэтому элемент b [1, 1] задает элемент во второй строке и втором столбце. Такая политика требует внимательности при работе с индексами.

В объявление типа массива границы не входят. Массивам память отводится динамически (как в Eiffel). Присваивание:

```
a = new string[4];
```

создаст массив из четырех элементов, инициализируемых стандартными значениями по умолчанию. Разрешается инициализировать массив значениями в момент его создания:

```
a = new string [] {"A", "B", "C", "D"}; //Массив из четырех элементов
```

При инициализации многомерных массивов используются запятые, разделяющие списки, которые заключены в фигурные скобки:

```
b = new string[2, 3];
b = new string[,] {{"A", "B"}, {"C", "D"}, {"E", "F"}}; // Размерность [3,2]
```

Кроме прямоугольных массивов, C# предлагает изрезанные, гребенчатые массивы, которые являются массивами массивов, как в примере:

```
string[][] c;
```

Каждая строка может иметь различный размер (отсюда и название – изрезанность, гребенка). Вот пример типичной инициализации:

```
c = new string[][] {new string[] {"A"},
                          new string[] {"B", "C", "D"},
                          new string[] {"E", "F"}};
```

Число элементов в каждой строке соответственно – 1, 3, 2.

Доступ и модификация использует синтаксис с квадратными скобками:

```
b[0, 0] = "Z";
c [0] = new string[] {"Y", "Z"};    // Изменяет первую строку с индексом 0
c [0][0] = "Z";
```

Можно определить нотацию со скобками для доступа к структурам, отличным от массивов, как в следующем примере:

```
Table t = new Table (); string n;
…
n = t [1, 1];    // Доступ к первому элементу (смотри реализацию ниже)
```

Здесь Table – собственный класс, поставляемый с индексатором, выполняющим ту же роль, что и псевдоним (alias) «[]» – «квадратные скобки» в Eiffel. Определение индексатора является обобщением метода-свойства:

```
class Table {
    private string[ , ] rep;// Инициализация rep опущена
    public string this [int i, int j] {
        get {return rep [i - 1, j - 1];}
        set {rep [i - 1, j - 1] = value;}
    }
}
```

Имя индексатора фиксировано и совпадает с именем текущего объекта this, как следствие, у класса может быть только один индексатор. Индексатор определяет два метода – геттер и сеттер с аргументами, задающими индексы элемента контейнера, к которому осуществляется доступ. В реализации класса Table индексация идет по двумерному массиву rep и устроена так, что для клиента начальный индекс по обоим измерениям начинается с единицы.

Универсальность

Концепция универсальности C# знакома по Eiffel; родовые параметры заключаются в угловые скобки <...>. Объявим:

```
class F<G, H> where H: T, new() {
    … Объявление класса …
}
```

Класс F имеет два родовых параметра. Для H задано ограничение (как в классе C [G, H –> T] в Eiffel), так что любой фактический родовой параметр должен наследовать от T. Включение в ограничение new() означает (как в Eiffel, если объявление имеет вид *C* [*G, H* –> *T* **create** *make* **end**] для процедуры создания *make* из *T*), что *T* должен обеспечить public-конструктор без аргументов; это позволит методам F создавать экземпляры T.

Родовое порождение также использует угловые скобки:

```
F<V, W>   //W должен быть согласован с T и иметь конструктор без аргументов.
```

Основные операторы

Дадим обзор основных операторов языка.

Присваивание, уже появляющееся в примерах, использует знак равенства:

```
var = e;
```

Заметьте: точка с запятой является не разделителем операторов, а завершителем, и должна завершать любой оператор.

Вызов метода использует имя метода и список аргументов. В отличие от вызова в Eiffel, круглые скобки всегда сопровождают вызов метода, даже если список аргументов пуст, как в methodWithNoArgument ().

Оператор return не имеет прямого аналога в Eiffel. Он обязателен для функций, задавая значение, возвращаемое функцией:

```
return some_expression;
```

Для процедур оператор возможен и задает завершение процедуры. Поскольку в методах он может появляться в нескольких местах, это означает, что методы C# не соответствуют правилу «один выход».

Управляющие структуры

Условный оператор удовлетворяет следующему синтаксису:

```
if (c1) {
    …
} else if (c2) {
    …
} else {
    …
}
```

Булевские выражения заключаются в круглые скобки. Отступы позволяют отражать структуру вложенности.

Множественный выбор имеет форму:

```
switch (expression) {
    case value: statement; break;
    case value: statement; break;
    …
    default: statement; break
}
```

Здесь *expression* задается булевским или целочисленным выражением, а каждое *value* представляет вычислимую в период компиляции константу. Когда значение выражения не совпадает ни с одной константой, выполняется ветвь default, если она задана, в противном случае ничего не делается (в Eiffel в отсутствие ветви **else** в операторе **inspect** в подобной ситуации в период выполнения генерируется ошибка). Оператор switch не задает конструкцию с одним входом и одним выходом, а представляет многоцелевой **goto**. Для правильной структурированности следует четко следовать показанной схеме. Принудительный оператор break, завершающий каждую ветвь, позволяет избежать типичной ошибки для C++ и C-версии switch, когда управление проваливается в другую ветвь.

Доступно несколько форм оператора **цикла**. Наиболее общая идет от C и соотносится с конструкцией **from** в Eiffel:

```
for (initialization; exit; modification) {
… body …
}
```

Вначале выполняется инициализация цикла – *initialization* – и работа цикла заканчивается, если условие выхода из цикла – *exit* – получает в результате значение *true*. Обычно оно не выполняется после инициализации, и тогда выполняется тело цикла, а затем модификация параметров цикла – *modification*, после чего снова проверяется условие выхода. Цикл завершается, когда выполняется условие выхода. Модификация обеспечивает продвижение к следующему шагу, увеличивая индекс, продвигая курсор (в Eiffel она интегрирована с телом цикла).

Можно использовать циклы в стиле while или until:

```
while (condition) {statements}
do {statements} while (condition)
```

Удивительно, C# сохранил оператор goto *M*, где *M* – метка. Операторы можно снабжать метками, отделяя метку от оператора двоеточием.

Обработка исключений

Исключение – это событие периода выполнения программы, появление которого приводит к прерыванию нормального выполнения программы. Причины исключений могут быть разные, такие как деление целого на нуль, отсутствие файла, null-ссылка у цели, вызывающей метод. Возможно также, что исключение «выбрасывается» при выполнении специального оператора языка C#:

```
throw e;
```

Здесь e – тип исключения, который должен быть потомком библиотечного класса Exception.

Обработка исключений в C# выполняется в следующем стиле:

```
try {
    … Обычные операторы, во время выполнения которых может возникнуть
    … исключение …
} catch (ET1 e) {
    … Обработка исключения типа ET1, детали в объекте e.…
} catch (ET2 e) {
    Обработка исключения типа ET2, детали в объекте e.…
}… Возможно, другие случаи …
finally {
    … Выполняется во всех случаях, было исключение или нет.
}
```

Если в охраняемом try-блоке включается исключение одного из перечисленных типов ET1, ET2, ..., то выполнение в try-блоке прерывается и управление передается соответствующему catch-блоку, который в состоянии захватить исключение данного типа. Блок finally выполняется всегда, если присутствует. Его обычная цель – освобождение ресурсов, закрытие файлов и так далее.

Появление исключения создает объект исключения, доступный программе в соответствующем catch-блоке. Это дает возможность при обработке исключения использовать свойства объекта, такие как имя исключения, понятное человеку, состояние стека вызовов.

Если встретилось исключение, чей тип не совпадает с типами, перечисленными в catch-блоках, или исключение встретилось вне охраняемого try-блока, то оно передается вверх по цепочке вызовов – в вызывающий метод. Здесь, опять-таки рекурсивно, исключение может быть обработано, если предусмотрен catch-блок, или передано наверх. При завершении цепочки вызовов, если обработка исключения не выполнена, то программа завершается стандартным сообщением об ошибке – обрабатывается стандартным catch-обработчиком исключения.

Читатель, знакомый с Java, заметил, что приведенное описание применимо к обоим языкам (получившим этот механизм от C++). Есть отличие от модели Java – метод в C# не специфицирует типы выбрасываемых исключений, как это делает Java, используя спецификацию throws.

Делегаты и события

В C# предлагается механизм делегатов (аналог агентов Eiffel) для описания методов как объектов. Ассоциированный механизм – события – дополняет делегаты для программирования, управляемого событиями (в Eiffel типы событий описываются как обычные объекты и, следовательно, нет необходимости в специальных конструкциях).

Рассмотрим объявление делегата:

```
public delegate int DT (string s);                                    [4]
```

Это объявление типа: оно определяет тип DT, который представляет функции с одним строковым аргументом, возвращающие целое в качестве результата. Для определения экземпляра класса и связывания его с конкретным методом – скажем, int lettercount(string s) – функцией, подсчитывающей число буквенных символов, – можно использовать:

```
DT d = new DT(lettercount)                                            [5]
```

Можно обойтись без вызова конструктора, используя явное присваивание:

```
DT d = lettercount;                                                    [6]
```

Некоторые языки программирования (отличные от функциональных) не позволяют использовать метод как аргумент другого метода. Механизмы агентов, делегатов, указателей функций в C++ спроектированы как раз с целью создания специальных объектов, передаваемых как аргументы и представляющих обертки соответствующих функций. Для создания в C#делегата из метода необходимо передать метод конструктору, как в **[5].** Концептуально это единственный случай, допускающий использование имени метода как значения.
На практике C# ослабляет правило, допуская присваивания, такие как **[6],** или передавая имя метода как аргумент другому методу, но это синтаксический сахар, фактически передаваемое значение является делегатом, здесь new DT ( ).

Делегат можно вызывать подобно любому другому методу:

```
n = d ("A");                                                           [7]
```

После присваивания **[5]** или **[6]** эффект будет тот же, как и при непосредственном вызове n = lettercount («A»);. Конечно, при вызове **[7]** обычно неизвестно, какую именно функцию представляет d. Часто такой вызов осуществляется в методе r, для которого d – формальный аргумент, и вместо присваивания, такого как **[6]**, передается делегат в качестве фактического аргумента:

```
r (lettercount);
```

Эквивалент Eiffel комбинации **[4]** или **[5]** задается одним оператором *d:=* **agent** *lettercount*, не требующим объявления типа, такого как **[4].** Эта форма не имеет прямого эквивалента в C#, но можно использовать делегаты с «анонимными методами» (эквивалент встроенных агентов), как в примере:

```
r (delegate (string s) {return lettercount (s);} );
```

Здесь используется анонимный метод. Заметьте, что анонимный метод объявляет сигнатуру, но не объявляет тип результата. Хотя в C# нет прямого эквивалента Eiffel-понятия «открытый аргумент», но анонимные методы позволяют достичь того же результата.

Эквивалентом встроенного агента Eiffel является лямбда-выражение C#, как в

```
(int x, int y) => x + y
```

соответствующее математической записи: $\lambda\, x, y: INTEGER \mid x + y$.

В C# делегат не ограничен представлением ровно одного метода. Если a и b – делегаты одного типа, то a + b обозначает делегата того же типа. Его выполнение приводит к последовательному выполнению методов, связанных с a и b. Операции add и remove позволяют в список методов, связанных с делегатом, добавлять или удалять новые члены. Можно для этого использовать и операцию присваивания += и –=.

Для программирования, управляемого событиями, можно определить типы событий и связать их с делегатами, как в следующем объявлении:

```
public event DT1 click;
```

Здесь click определяется как событие, которое будет обрабатываться делегатом типа DT1.

Этот пример использует новый тип делегата DT1 вместо рассматриваемого ранее DT, так как типы делегатов, обрабатывающих события, обычно являются процедурами, в то время как DT задает класс функций. Если мы собираемся рассматривать щелчки кнопки мыши, то надо передать событию аргументы, задающие координаты мыши, тогда DT1 будет объявлен как:

```
public delegate void DT1 (int x, int y);
```

Для понимания механизма необходимо знать, что реализация C# представляет каждый тип события, такой как click, как список делегатов, которые соответствуют различным методам, подписанным на события. Это объясняет, как подписаться на события:

```
click += r;
```

Здесь r – это метод с подходящей сигнатурой: void r (int x, int y). Операция += перегружена для списков; она добавляет элемент в список. Заметьте, что можно применить метод r непосредственно, прежде чем явно обернуть его в одежды делегата. Но то, что получается при добавлении в список, является делегатом. Для удаления подписки используется конструкция –=.

Так выполняется подписка. Для публикации события используется схема:

```
if (click != null) {click (h, v);}
```

В этом примере *h* и *v* задают координаты мыши. Опять-таки, нужно знать о реализации списка, чтобы понимать необходимость теста *click* != **null**: если нет делегатов, подписанных на click, то список будет иметь значение **null**, и вызов *click* (*h*, *v*) станет причиной исключительной ситуации.

Рекомендуемый стиль для обработки аргументов события в .NET состоит в том, чтобы аргумент события был объявлен как потомок библиотечного класса EventArgs. В этом примере можно было бы объявить класс IntPairArgs, наследуемый от EventArgs, и в этом классе объявить два целочисленных поля x и y. Тогда подписываемые методы имели бы форму:

```
private void rHandler (object sender, IntPairArgs e) {r (e.x, e.y);}
```

Здесь – как часть того же рекомендуемого стиля – первый аргумент sender представляет целевой объект, второй аргумент e представляет аргументы события. Преимущество в том, что все схемы обработки события выглядят одинаково. Неясно, однако, оправдывает ли это возникающие усложнения: вместо прямого повторного использования методов, существующих в модели, таких как r, необходимо обернуть их в специальный склеивающий код, такой как rHandler.

В.4. Наследование

Модель наследования C#, совпадающая в основном с моделью Java, не поддерживает множественного наследования классов, за исключением специального вида абстракции, называемого интерфейсом.

Наследование от класса

Вот как класс объявляет другой класс своим родителем:

```
class L :K
{… Объявления компонентов L …}
```

Класс L объявил себя наследником K. Следующий пример показывает случай, когда наследуемый класс объявляется универсальным, здесь – с ограниченной универсальностью:

```
class M<G>:K where G: T
{… Объявления компонентов M …}
```

Только один класс, здесь – K, может быть объявлен в качестве родителя.

Абстрактные методы и классы

В C# можно объявлять абстрактными как отдельные методы, так и класс в целом (это соответствует отложенным методам и классам Eiffel, но без контрактов):

```
abstract class N {
    public abstract void r(); // Заметьте, реализация отсутствует
    public abstract int s(); // Заметьте, реализация отсутствует {…}
    … Другие методы, которые могут быть или не быть абстрактными…
}
```

Нельзя использовать конструктор для создания экземпляров абстрактного класса (как new N (...)). Абстрактный метод может появиться только в абстрактном классе. Неабстрактные потомки должны обеспечить переопределение абстрактных методов родителя. Нельзя одновременно объявить метод класса абстрактным и статическим.

Методы-свойства и индексаторы также могут быть абстрактными.

Интерфейсы

Интерфейс подобен полностью абстрактному классу. Интерфейсы задают абстрактную функциональность, которую каждый потомок должен реализовать. Интерфейсы обеспечивают ограниченную форму множественного наследования, так как каждый класс может наследовать от множества интерфейсов. Это позволяет языку избегать конфликтов, когда несколько наследуемых методов от разных родителей с одинаковыми именами имеют разные реализации (мы знаем, что конфликты разрешимы за счет переименования). Платой является некоторая потеря мощности наследования.

Пример типичного интерфейса:

```
interface IOrdered <T> {
    bool lesser (T other);
    bool greater (T other);
    bool lesserEqual (T other);
    bool greaterEqual (T other);
}
```

По соглашению имена интерфейсов должны начинаться с буквы I. Пример показывает, что интерфейс может быть универсальным. Важно, что интерфейсы объявляются без указания модификатора доступа, – они неявно имеют статус public[1].

> Данный интерфейс задает свойство сравнимости объектов, используя общепринятые имена операций сравнения. Он также иллюстрирует границы понятия интерфейса – поскольку методы не могут иметь реализации, нельзя задать, что a.greater (b) должно быть реализовано как b.lesser (a) (для абстрактного класса такое возможно, но только один родитель может быть классом, не имеет значения – абстрактным или эффективным).

Класс может наследовать от одного или нескольких интерфейсов, обеспечивая реализации методов:

```
class TennisPlayer: IOrdered<TennisPlayer>, IAnotherInterface {
    public int ranking;
    public bool lesser (TennisPlayer other) {return (ranking < other.ranking);}
    … Аналогично для greater, lesserEqual, greaterEqual …
    … Реализация методов IAnotherInterfacei …
    … Другие компоненты TennisPlayer …
}
```

Множественное наследование интерфейсов не исключает конфликта имен. В C# нет явного механизма переименования, аналогичного Eiffel, но конфликта можно избежать, используя в качестве префикса имя интерфейса и нотацию с точкой, например, IAnotherInterface.clashingname.

Доступность и наследование

В связи с наследованием в дополнение к трем ранее рассмотренным модификаторам доступа (public, internal, private) добавляется новый модификатор и новая комбинация:

- protected: доступно для потомков;
- protected internal: доступно потомкам и классам той же сборки.

Когда переопределяется наследуемый компонент, как обсуждается далее, статус доступа изменить нельзя. Этим язык отличается от Java, где разрешается расширить права доступа, но не ограничить их. Кроме того, потомки не могут иметь доступность большую, чем их предки.

Переопределение и динамическое связывание

Наследуемый метод можно переопределить. Соглашения, однако, отличаются от других современных языков программирования. Как в C++, и в отличие от Eiffel и Java, связывание в C# – удивительно для языка, первая версия которого появилась в 1999 году, – статическое. Другими словами, версия f, выполняемая при вызове a.f, будет следовать объявлению a, но не динамическому типу объекта, связанному с a в момент выполнения. Чтобы выполнялось динамическое связывание, метод должен быть объявлен как виртуальный – virtual[2]:

---

[1] Причина не только в этом. Потомку дается свобода. Потомок может реализовать интерфейс как открытый или как закрытый. Это крайне важно при конфликте имен. Первый случай реализует склеивание одноименных методов, второй – дает возможность обертки и переименования.

[2] Для абстрактных методов также применяется динамическое связывание. Они ведут себя в этом отношении как виртуальные методы.

```
class P {
    public virtual void f (…) {…}
    …
}
```

Потомок переопределяет виртуальный метод, в точности сохраняя сигнатуру и заменяя модификатор virtual на override:

```
class Q: P {
    public override void f (…) {…}
    …
}
```

Оригинальная версия переопределяемого метода остается доступной, как с **Precursor** в Eiffel, для этого применяется нотация с точкой, а в качестве префикса используется имя родителя – base. Например, реализация f в Q может пользоваться результатами работы, проделанной родителем:

```
base.f (n);
```

Для реализации динамического связывания требуется точное соответствие схеме (оригинал метода специфицирован как virtual, новый – как override[1]). Если условие не выполняется, то компилятор по-другому интерпретирует текст. Рассмотрим следующий вариант:

```
class R {public void f (int i) {…}}
class S: R {public void f (int i) {…}}
R r1 ; S s1 = new S() ; int n;
r1 = s1;
```

Оригинал f не объявлен virtual в R, но все же потомок S дал новую реализацию. Это не является переопределением метода родителя. Это интерпретируется как создание потомком нового метода с тем же именем, скрывающего метод родителя (доступный через base). В этой ситуации применяется статическое связывание. Вызов s1.f(n) будет использовать новую версию, но вызов r1.f(n) будет использовать версию R независимо от того, с каким объектом связано будет r1 во время выполнения. Это довольно опасно, хотя компилятор отслеживает ситуацию и выдает предупреждающие сообщения. Для защиты от риска ошибки новый метод следует помечать как new:

```
class S: R {public new void f (int i) {…}
```

Динамическое связывание не встретится и в случае вызова p1.f (n), где p1 типа P, и динамически типа Q, если в Q опустить при переопределении модификатор override. Это не будет восприниматься как ошибка, просто означает применение статического связывания.

При переопределении можно задать модификатор sealed, чтобы защитить метод от переопределения у потомков:

[1] Метод родителя может также иметь модификаторы abstract или override.

```
class Q1: P {
    public sealed override void f (…) {…}
    …
}
```

Этот модификатор допускается только для override-методов, поскольку остальные по определению не являются переопределяемыми. Модификатор sealed может быть у класса (sealed class T ...), и тогда такой класс не может иметь потомков.

> Объявление классов и методов с модификатором sealed характерно для .NET-библиотек, вероятно, по той причине, что в условиях отсутствия контрактов это единственный способ не позволить потомкам исказить намерения родителя.

Выбор в языке C# по умолчанию статического связывания ошибочен. Из этого наблюдения следует методологическое правило для C#-программистов: *всегда объявлять методы* **virtual**, удаляя этот модификатор в тех редких случаях, когда метод должен быть защищён от переопределения – sealed.

Наследование и создание

В C# в присутствии наследования действует специальное правило для конструкторов: любой конструктор должен вызвать конструктор родителя. Как результат, вызов конструктора приводит к цепочке вызовов, включающей вызовы всех родительских конструкторов и заканчивающейся вызовом конструктора по умолчанию прародителя – object. Этот эффект может достигаться явно или неявно.

- Конструктор может вызвать конструктор родителя, используя нотацию base (как base (n);). Так как конструкторы не имеют индивидуального имени, сигнатура аргументов, благодаря правилам перегрузки, однозначно определяет, какой конструктор будет вызван.
- В отсутствие такого вызова родитель должен иметь конструктор без аргументов, который будет автоматически выполняться перед тем, как начнет свое выполнение конструктор потомка.

Один из этих случаев должен быть применим. Если в конструкторе нет явного вызова конструктора родителя и родитель при этом не имеет конструктора по умолчанию, то возникнет ошибка периода компиляции.

Как отмечалось при обсуждении соответствующих механизмов Java, причина этих правил не полностью ясна. Намерением, вероятно, было убедиться, что экземпляр потомка удовлетворяет правилам согласованности, определенными предками. Цепь конструкторов появляется как попытка убедиться в такой согласованности. В отсутствие понятия инварианта класса она позволяет явно выразить ограничения.

Идентификация типа в период выполнения

Для приведения к типу U выражения exp типа T (как в тесте объектов Eiffel) можно использовать два механизма.

- Явный кастинг – написать (U) exp, не учитывая объявленный тип exp. Если все хорошо и exp в самом деле представляет объект типа U в момент выполнения, то можно использовать это выражение для ссылки на значение приведенного типа. Обратная сторона в том, что если динамический тип не соответствует U, попытка вычисления выражения приведет к возникновению исключительной ситуации, обработку которой надо предусмотреть для безопасного программирования.

- Более приемлемым способом является использование специальной C#-конструкции – булевского выражения: exp is U.

Во втором случае *exp* по-прежнему статически принадлежит объявленному типу *T*, но теперь можно комбинировать два механизма с гарантией, что кастинг будет работать:

```
if (exp is U) {r ((U) exp)};     // Метод r объявлен как: r(U : x) {…}
```

B.5. Другие механизмы структурирования программ

В C# введены несколько механизмов структурирования программ, не входящих в базисную ОО-парадигму.

Пространства имен

При конструировании ПО, приходящего из разных источников, возможны конфликты имен используемых классов: два провайдера могут поставлять классы с одинаковыми именами. В C# такие конфликты могут разрешаться благодаря введению уровня структуризации более высокого, чем классы, – пространства имен.

По умолчанию имена всех типов (классов) принадлежат глобальному пространству имён. Разрешается определять собственное пространство имен, содержащее объявления классов:

```
namespace N1 {
… Объявления классов (и других типов по желанию) …
}
```

В этом случае клиент, которому необходим доступ к нескольким классам с одним именем, скажем, *C*, может устранить неопределенность, используя нотацию N1.C.

Важным предопределенным пространством имен является пространство *System*, содержащее базисные библиотеки классов.

Пространства имен могут быть вложенными. Это достигается как прямой вложенностью текста одного пространства в другое, так и с использованием нотации с точкой:

```
namespace N1.N2 {// N2 - подпространство N1: элементы доступны в нотации
                  // N1.N2.V.
… Объявления классов, включая V …
}
```

Клиент, часто использующий классы из некоторого пространства имен, может избежать повторения квалифицированных имен (как в N1.N2.C, N1.N2.D и т. д.) благодаря использованию директивы using:

```
using N1.N2;
using SomeOtherNamespace;
… Здесь можно использовать V как сокращенную запись N1.N2.V …
```

Как показано в примере, можно использовать любое число ***using***-директив. Если соответствующие пространства имен содержат классы с конфликтующими именами, то приходится вернуться к нотации с точкой для разрешения конфликтов.

Методы расширения

Предположим, вы хотите расширить концепцию, покрываемую существующим классом X. Обычный ОО-механизм состоит в объявлении нового класса, наследуемого от X. Но наследование может быть неудобным или неприменимым. Например, X может запрещать наследование (как отмечалось, обычная практика для .NET-библиотек). Вообще, определение нового класса означает определение нового типа. Даже если вы только добавляете методы, не добавляя полей, система типов не позволит применять новые операции к существующим объектам исходного типа X, например, к объектам, хранящимся в файле или в базе данных.

Чтобы справиться с этой проблемой, C# обеспечивает интересный механизм расширения методов: методы добавляются извне к существующему классу.

Очевидно, методы расширения представляют просто синтаксическое упрощение, так как проблему можно решить за счет статических методов: в любой класс можно добавить статический метод sm, который будет вызываться как sm (x1, other_args) с x1 типа X. Однако мы хотим, чтобы метод m вызывался в том же стиле, как если бы он был методом X:

```
x1.m (other_args);                                                  [9]
```

При этом m объявляется не в классе X, а в другом классе. Синтаксический трюк состоит в том, чтобы маркировать первый аргумент m модификатором this:

```
public static class Y {
    static void m (this X x, int arg1, int arg2) {…}
    …
}
```

В результате в классе X появляется расширенный метод и **[9]** становится правильной конструкцией (с *двумя* целочисленными аргументами, заменяющими other_args).

Атрибуты

По своей природе языки программирования ограничены семантическими механизмами, предусмотренными их создателями. Иногда возникает желание добавить новые свойства, не изменяющие существующую семантику, но полезные, например, для документирования или сериализации.

Вполне разумно для языка программирования обеспечить возможность таких расширений, известных как включение метаданных, — поддерживающей информации, добавляемой в документ, отделенной от основного содержания. В Eiffel именно для такой цели используются предложения **note**, связанные с классами и методами. В .NET и C# поддержка метаданных обеспечивается в форме атрибутов (не путайте с ОО-понятием атрибута, используемым в этой книге, для которого в C# применяется термин «поле»).

Некоторые атрибуты предопределены, но программисты могут определять собственные атрибуты, известные как атрибуты пользователя.

Примером предопределенного атрибута является атрибут сериализации Serializable, который можно присоединить к классу, указав тем самым, что экземпляры могут быть конвертированы в некоторое внешнее представление и храниться во внешней памяти:

```
[Serializable]
public class Z {… Обычное объявление класса …}
```

Здесь показано добавление атрибута к классу. То же соглашение действует при связывании атрибута с методом. Атрибут заключается в квадратные скобки и используется в качестве префикса класса или метода.

Для задания пользовательского атрибута следует определить класс – потомок класса System.Attribute (то есть класса Attribute из пространства имен System). Предположим, что мы хотим поставлять классы с базисной версией управляющей информации: именем автора, датой модификации, возможно, номером версии, все в строковом формате. Мы используем:

```
public class ChangeAttribute: System.Attribute {
    private string author;
    private string last;
    public ChangeAttribute (string a, string l)
       {author = a ; last = l;}
    public string revision;
}
```

Заметьте, имя класса заканчивается словом Attribute, – это рекомендуемый стиль именования атрибутных классов.

Тогда можно поставлять класс или (здесь) метод с информацией о версии:

```
[ChangeAttribute ("Caroline", "24 December 2009")] public void r {…}
```

При задании атрибута мы должны передать аргументы выбранному конструктору. Также возможно задать значения public полей атрибутного класса, так называемых полей, допускающих запись. В данном примере таким является поле revision:

```
[ChangeAttribute ("Caroline", "24 December 2009", revision = "2.1" )]
public void r {…}
```

Атрибут ChangeAttribute, в том виде как он объявлен, применим к любым программным элементам: class, struct, method, field, delegate и некоторым другим. Можно ограничить применимость атрибута, присоединив собственное объявление у атрибута AttributeUsage:

```
[System.AttributeUsage(System.AttributeTargets.Class)]
public class ChangeAttribute: System.Attribute {… Остальное, как прежде …}
```

Вместо задания Class (в качестве области действия атрибута) можно использовать другие ключевые слова: All (по умолчанию), Assembly, Delegate, Event, Interface, Field, Method, Parameter, Struct. Разрешается задавать несколько целей, разделяя их символом вертикальной черты |.

Для элементов, снабженных атрибутами, можно получать значения атрибутов через процесс, называемый отражением. Рассмотрим вызов объекта o:

```
o.GetType().GetCustomAttributes(true);
```

Результатом будет массив, содержащий атрибуты, определенные для класса объекта, с их значениями.[1]

[1] Боюсь, что для тех, кто не знаком с атрибутами C#, этот краткий обзор недостаточно информативен. По крайней мере, он вводит некоторые понятия, с которыми при желании можно ознакомиться подробнее.

В.6. Отсутствующие элементы

В C# отсутствуют несколько ОО-механизмов, интенсивно применяемых в этой книге, в первую очередь – контракты и множественное наследование.

Контракты введены в исследовательскую версию языка C# – SpecC#, разработанную в Microsoft Research. Эта версия языка доступна для свободного использования.

В.7. Специфические свойства языка

Полезно ознакомиться с несколькими поддерживающими C# конструкциями.

Небезопасный код

В C# комбинируются строгие требования к безопасности типов с предоставлением программистам возможности работы на низком уровне, характерном для языка C или ассемблера. Конструкция «unsafe» поддерживает четкое разделение между небезопасными элементами и нормальными, прошедшими проверку типов.

Объявление метода небезопасным (unsafe) означает, что область данных расположена вне «кучи» – области, отводимой под объекты. Небезопасный метод может осуществлять в своей области непосредственные манипуляции с указателями и, следовательно, игнорировать нормальные правила работы с данными.

Тип «перечисление»

Типы, заданные перечислением, позволяют оперировать значениями из конечного множества предопределенных значений:

```
enum CardColors {Spades, Hearts, Diamonds, Clubs}
```

Значения, заданные перечислением, проецируются на целочисленный тип, по умолчанию int, но можно задать и другой тип, как в enum T:long {...}. Значения проецируются на отрезок, начинающийся с 0, но можно задать и другое начало:

```
enum CardColors1 {Spades = 1, Hearts, Diamonds, Clubs}
```

Значения можно обозначать, используя нотацию с точкой, как в CardColors.Spades. Допустимы взаимные преобразования между целыми и значениями перечисления.[1]

Linq

В версии языка C# 3.0 появились новые важные механизмы, привлекшие внимание программистов. Механизм, известный как Linq, позволяет языку программирования работать непосредственно с базами данных и Web. Для работы с данными используется типичный язык запросов SQL и XML для Web. Обычно такие возможности поддерживаются с помощью специальных библиотек, обеспечивающих интерфейс к реляционным базам данных или к прото-

[1] Наиболее интересное применение перечислений – это шкалы. В этом случае каждый к-й элемент перечисления проецируется на значение 2к. Тогда переменная типа «перечисление» интерпретируется как набор битов, над которыми определены, как мы знаем, логические операции. Если необходимо работать с множеством объектов, характеризуемых набором бинарных свойств (свойство присутствует у объекта или нет; например, знает данный программист ОО-концепции или нет), то шкалы – прекрасный инструмент для таких задач.

колу HTTP. Оригинальность Linq в том, что он делает все гораздо выразительнее, оставаясь в рамках языка программирования. Например, запрос к базе данных можно выполнить так:

```
from e in Employees where e.salary > median select e.rank
```

Здесь идет ссылка на отношение Employees из базы данных. В результате создается список всех служащих, ранжированных по зарплате. Стиль тот же, что и у SQL-запросов, но запрос интегрирован в язык. Все используемые объекты – это нормальные объекты программы, такие как списки.

В.8. Лексические аспекты

В C# идентификаторы следуют соглашениям, подобным Eiffel. Они могут, однако, начинаться с подчеркивания, хотя эта возможность редко используется в обычных приложениях. Важная разница в том, что идентификаторы чувствительны к регистру: anIdentifier, AnIdentifier и anidentifier – это все различные идентификаторы. Лежащее в основе множество символов – это Unicode.

Комментарии в C# программах могут быть:

- однострочными: любая часть строки, начинающаяся с //;
- многострочными: начинающиеся и заканчивающиеся парой символов /* ...*/.

Однострочные комментарии, начинающиеся с /// (с добавленным слеш-символом), задают документированный комментарий, записываемый в виде XML-кода. Эти комментарии используются специальным инструментарием для разных целей – документирования, интеллектуальной подсказки.

В.9. Библиография

Judith Bishop, Nigel Horspool, C# Concisely, Addison-Wesley, 2003.

Введение в основные механизмы C# (для ранних версий языка).

Онлайн документация на сайте:
msdn.microsoft.com/en-us/vcsharp/default.aspx.

Лучшее место для получения детальной спецификации механизмов языка. Сайт также включает ссылки на несколько учебников.

На русском языке: Биллиг В.А. «Основы объектного программирования на C# 3.0», Интернет-Университет. Питер, Лаборатория знаний, 2010.[1]

[1] Не могу удержаться от рекламирования собственного учебника, тем более, что он соответствует по духу этой книге.

C
Введение в C++ (по материалам Надежды Поликарповой)

Общепринято характеризовать ОО-языки как «чистые» (полностью и исключительно реализующие ОО-концепции) либо как «гибридные» (представляющие смесь объектных и необъектных свойств). Наиболее известным представителем гибридных языков является язык C++, который создавался с целью предоставления C-программистам некоторых ОО-идей и механизмов. Язык C++ по духу отличается от нотации Eiffel, принятой в остальной части книги, и более сложен.

Можно найти много книг и вводных статей о C++, как начинающихся с чистого листа, так и предполагающих знание языка C. Роль этого приложения другая: она ориентирована именно на вас, внимательных читателей этой книги, кто прочел уже сотни страниц и овладел ОО-программированием в его «чистой» форме. Цель в том, чтобы в случае необходимости программирования на C++ применять его конструкции в духе Eiffel. По этой причине многие конструкции C++ будут объясняться – как это сделано для Java и C# — в стиле: «Вот как это можно сделать на Eiffel и как получить такой же эффект на C++».

Язык C, на котором основан C++, сам является важным языком, кратко рассматриваемым в следующем приложении.

C.1. Основы языка и стиль

Сегодня идея использования ОО-языка едва ли может удивить кого-либо, но в конце восьмидесятых годов она воспринималась как насмешка, — многим программистам и менеджерам в индустрии и академических кругах ОО-концепции казались привлекательными, но возникали большие сомнения в их применимости. В 1979 году Бьёрн Страструп из Bell

Рис. C.1. Бьёрн Страуструп (2007)

Laboratories спроектировал и реализовал язык, изначально названный «С с классами», расширяющий язык С концепциями, которые были заимствованы у языка Simula 67, — первого ОО-языка. Код транслировался в чистый С препроцессором. Язык вскоре стал хитом и, как было обещано, облегчил переходный период для С-программистов.

В следующие два десятилетия язык интенсивно развивался за счет введения таких конструкций, как шаблоны (форма универсальности) и множественное наследование.

С.2. Общая организация программ

Для ОО-подхода программа представляет множество классов. В С++, сохраняющем гибридный дух, не настаивают на соблюдении этого правила. Программа может включать классы, но и другие элементы, не входящие в классы: функции (соответствующие методам, реализуемым либо в виде настоящих функций, либо в виде процедур), переменные, константы и типы, не заданные классами.

Примером независимой функции, появляющейся в каждой исполняемой программе, является функция, называемая main, которая определяет точку входа в программу. В Eiffel эту роль играет корневая процедура создания.

Программная система на С++ представляет скорее не множество классов, а множество единиц трансляции (юнитов или модулей), каждая из которых содержится в отдельном файле, который независимо может быть обработан С++ компилятором. Каждая единица трансляции может содержать объявления классов и других типов, функций, переменных и констант.

Объявление такого элемента может быть его определением, означающим его полное описание, либо может объявлением, не задающим определение, таким как:

```
class Person;
enum Week_day;
```

При этом понимается, что полное определение этого элемента (здесь класс и тип) появится либо в другом модуле, либо в той же единице, но позже. Это дает возможность использовать элемент без знания его детальных свойств. Для переменной или константы неопределяющее объявление использует ключевое слово extern, указывающее, что определение еще появится:

```
extern bool has_error;
extern const double pi;
```

Определение функции включает имя, сигнатуру и реализацию:

```
int factorial (int n)
    {
       if (n > 1) {
          return n _ factorial (n - 1);
       } else {
          return 1;
       }
    }
```

Определение класса содержит список членов класса (компонентов):

```
class Person { // Объекты, представляющие персон
    string name; // Имя персоны
    Date birth_date; // Дата рождения
    void set_name (string s)
       {name = s;}
    void set_birth_date (Date d)
       {birth_date = d;}
    … // Другие члены класса
};
```

В определении переменной задается ее тип, предшествующий переменной в отличие от Eiffel, где действует соглашение *variable_name*: *TYPE*.

```
int n;
bool has_error = false;
```

Второе из этих объявлений содержит инициализацию переменной, устанавливающую ее начальное значение. Определения не содержат extern: это означает, что переменные не являются внешними, а значит, могут использоваться только в данном модуле трансляции.

Определение переменной означает, что ей будет отведена память, как для развернутых типов Eiffel.

Определение константы всегда включает ее инициализацию:

```
const double Pi = 3.14159265358
```

В отличие от Eiffel порядок объявлений существенен: имена, объявленные в модуле трансляции, можно использовать только после точки объявления. Для применения элемента до его определения – такая необходимость возникает, например, в случае множества взаимно рекурсивных определений – предварительно следует задать неопределяющее объявление.

Во избежание трансляции слишком большого модуля можно разделить его на несколько файлов, используя затем директиву включения #include, как в примере:

```
#include "filename"
```

Как результат, все определения, содержащиеся в исходном файле filename, будут доступны текущему модулю трансляции. Чаще всего это свойство предполагает использование заголовочных файлов (по соглашению имеющих имена в форме *name*.h), которые содержат объявления элементов, используемых многими модулями, и включают соответствующую директиву #include.

Директива #include, подобно любым другим предложениям, начинающимся с символа решетки #, адресована препроцессору C++ – инструментарию, обрабатывающему файлы программы до начала компиляции. Другое использование препроцессора связано с условной компиляцией, позволяющей определять препроцессорные переменные и включать в выполнение некоторый код только при условии задания соответствующих переменных. Вот типичный пример:

```
#ifdef LINUX
    … Linux-specific code …
#endif
```

Код для платформы Linux будет подключен в зависимости от LINUX-переменной. Такие переменные, не связанные с переменными программы, рассматриваются как опции, действующие на этапе работы препроцессора и компилятора.

C.3. Базисная ОО-модель

Подобно Eiffel, каждая C++ переменная имеет тип, но, в отличие от Eiffel, не все типы основаны на классах, и следовательно, не все значения являются объектами. Тип может быть встроенным, производным (с возможными комбинациями механизмов порождения), определенным пользователем.

Встроенные типы

Встроенные типы предустановлены в языке. Они включают bool – для булевских значений; char, short int, int, long int – для целых из разных диапазонов; float, double, long double – для вещественных с плавающей точкой. Тип char также служит для представления символов.

Каждый целочисленный тип включает две версии – со знаком и без знака, такие как short int и unsigned short int. Версии без знака включают только положительные значения. По умолчанию все целые типы, за исключением char, являются знаковыми, делая избыточным задание такого типа, как signed int. Является ли char типом со знаком, определяется платформой.

Встроенный тип void не имеет значений. Он служит признаком процедур — функций, не возвращающих результат:

```
void set_name (string s);
```

Производные типы

Производные типы конструируются из уже существующих типов, используя одно из пяти возможных преобразований: в константу, указатель, ссылку, массив, функцию. Следующие примеры демонстрируют эти преобразования для получения нового типа из существующего типа T.

Тип const T представляет неизменяемое значение типа T. Например, n будет иметь тип const int, если ее определить как:

```
const int n = 5;
```

Разрешается переменной типа T присвоить значение типа const T, но обратное преобразование недопустимо.

Указателем на тип T является тип T*. Значения этого типа обозначают адреса памяти, где хранится переменная типа T. Для получения указателя на переменную *x* используется &, как в &x. Возможно обратное преобразование, называемое разыменованием – получением значения по адресу. Если p – указатель, то *p дает значение, хранящееся в области памяти, на

которую указывает указатель. Если значением является объект, то возможен доступ к его полям, например, полю f, используя нотацию с точкой: (*p).f. Специальный синтаксис p–>f является синонимом нотации с точкой.

Поскольку указатель представляет адрес памяти, в языке разрешено добавление или вычитание целого из указателя для получения нового адреса, как в*(p + n). Смысл его в следующем. Представим, что в памяти подряд хранятся n + 1 значение типа T, каждое занимает фиксированное число байтов, которое можно получить, используя конструкцию sizeof T. Тогда, если *p* типа *T** и указывает на первое хранимое значение, то *(p + n) возвращает последнее значение (здесь неявно n умножается на число байтов, отводимых элементу — sizeof T, так что (p + n) дает адрес начала соответствующего элемента, а операция * — значение по этому адресу). Это все называется адресной арифметикой и широко используется в низкоуровневых программах на C для получения доступа к нужным участкам памяти. Механизм мощный, но чреватый ошибками (трудно гарантировать, что на самом деле хранится в динамически вычисляемом адресе). Без необходимости его не следует применять в приложениях C++.

Все типы указателей согласуются со специальным типом void*, напоминающем класс *ANY* в Eiffel. Но это встроенное соответствие, не индуцированное наследованием. Поскольку нельзя иметь переменные типа void, нельзя проводить разыменование указателей void*, никакие операции над ними не выполняются. Чтобы их использовать, необходим кастинг – явное приведение к типу! В C++ есть разные способы приведения, дающие разные результаты в случае, когда операция приведения не заканчивается успехом.

Тип T& является «ссылкой на T». Подобно указателю, ссылка является адресом, но при любом использовании происходит автоматическое разыменование. Ссылки являются наиболее прямым способом получения эффекта обычного (ссылочного) класса в Eiffel. Рассмотрим класс Eiffel **class** *PERSON* ... **end**, и пусть в этом классе определен некоторый метод:

```
call_her_izzy (p: PERSON)
    do p.set_name ("izzy") end
```

Эквивалентом метода в C++ будет:

```
void call_her_izzy (Person& p)
    {p.set_name ("izzy");}
```

Рассмотрим вызов этого метода call_her_izzy (Isabelle), где Isabelle типа Person. Эффект состоит в изменении значения поля имени ссылочного объекта. Если бы аргумент в C++ версии имел тип Person, то вызов создавал бы копию объекта Person.

Метод работал бы на этой копии, но без видимого эффекта, так как копия локальна и исчезла бы по завершении выполнения метода.

Реализовать ссылочное поведение можно и с указателями, но сложнее — с явной адресацией и разыменованием:

```
void call_her_izzy (Person* p)
    {p->set_name ("izzy");        // или (_p).set_name ("izzy");
    }
```

В этом случае вызов метода имеет вид: call_her_izzy (&Isabelle). Использование указателей считается хорошим стилем в сравнении с передачей аргументов по ссылке, поскольку делает явной ссылочную семантику.

Еще одна разница между указателями и ссылками в том, что ссылки требуют инициализации, присоединяющей значение типа T к T& ссылке, в то время как указатели могут иметь нулевое значение (называемое также null и соответствующее void в Eiffel). Однако это не обеспечивает преимущества присоединенных типов Eiffel, так как возможно присвоить указатель T* переменной типа T& или T, что станет причиной ошибки периода выполнения, если указатель равен null.

В целом ссылочные типы обеспечивают более строгую дисциплину, в частности, они не позволяют адресную арифметику.

Теперь поговорим о массивах. Тип, задающий массив (иногда говорят «массивный тип»), — T[size], где размер size является целой константой, известной во время компиляции, представляет последовательность значений типа T, хранимой в подряд идущих словах памяти. Если ar — это массив, то ar[i] обозначает i-й элемент этого массива, где i — выражение целого типа. В C++ массив рассматривается как адрес начала расположения элементов массива в предположении, что памяти для их хранения достаточно. Для безопасной работы с индексами, гарантирующей, что выход за границы контролируется, следует использовать библиотечные классы.

Наконец, функциональный тип. Он фактически является указателем на тип функции. Пусть R, A1, ..., An являются типами (последовательность задает сигнатуру функции). Рассмотрим объявление:

```
R (*f) (A1, …, An);
```

Здесь объявлена переменная f, чьи значения являются указателями на функции, возвращающие значение типа *R* и имеющие *n* аргументов типа A1, ... An. Например, рассмотрим объявление:

```
void (*f) (Person*)
```

Переменной f можно присвоить указатель на функцию:

```
f = call_her_izzy;  //или f = &call_her_izzy;
```

После этого можно выполнять непрямой вызов:

```
f (Isabelle);       // или (*f) (Isabelle);
```

Эффект будет тот же, что и при прямом вызове call_her_izzy (Isabelle). Разница в том, что f — это переменная, которой можно присвоить указатели на разные функции (с заданной сигнатурой).

Указатели функций могут обозначать не только независимые функции, но и функции, представляющие членов класса, как в примере:

```
void (Person::*p) (string)
…
p = Person::set_name;
```

Здесь объявляется p и ему присваивается указатель на функцию класса Person, принимающий один аргумент типа string. Эту функцию можно вызывать, используя следующий синтаксис:

```
(Isabelle.*p) ("Izzy");
```

Как вы уже понимаете, указатели на функции близко связаны с двумя ОО-механизмами, основанными на способности вызывать метод, оставляя на момент выполнения определение того, какой именно метод будет вызван. Этими механизмами являются динамическое связывание и агенты. Указатели функций С++ дают возможность эмуляции этих свойств.

- Они являются лучшим способом получения эффекта агентов, хотя и не в полной мере, поскольку агент — это объект со многими свойствами и гарантией типизации, а все, что можно делать с указателем, сводится к вызову функции.
- Указатели функций позволяют получить эффект динамического связывания, используя динамический тип объекта как индекс в массиве указателей функций, позволяющий выбрать правильную версию функции. Эта техника детально была описана при обсуждении реализации наследования. Нет необходимости в ее использовании, поскольку С++ предоставляет механизм виртуальных функций, изучаемый ниже, который реализует динамическое связывание. Поскольку в чистом С виртуальных функций нет, у них остается только одна возможность — применение указателей функций. Вот почему компиляторы Eiffel, транслирующие программу в код на С, основываются на объясненной ранее схеме.

Комбинирование механизмов производных типов

Пять механизмов построения производных типов могут комбинироваться различными способами. Вот примеры некоторых важных комбинаций.

Можно комбинировать модификаторы const и pointer, чтобы получить:

- константный указатель, который всегда будет указывать на одну и ту же область памяти, хотя значение, хранимое в этой области, может меняться;
- указатель на константу, который может быть изменен, но не может использоваться для изменения значения в области, на которую он указывает (но на эту область могут указывать другие, не константные указатели, позволяющие изменить значение);
- константный указатель на константу, для которого и область, и сам указатель неизменяемы.

Подобным образом можно комбинировать const и ссылки. Следующие примеры иллюстрируют некоторые из возможностей:

```
const int* pointer_to_const;
const int& reference_to_const;
int* const const_pointer;
int& const const_reference;
```

На практике важной является нотация typedef, позволяющая именовать создаваемые типы. Это позволяет ссылаться на сложный производный тип, используя его простое и понятное имя:

```
typedef const Person* Cp;
typedef void (* Pf) (Person*);
```

Теперь возможно использовать в объявлениях Cp p вместо const Person *p. Тип Pf обозначает теперь соответствующий тип указателя на функцию.

Типы, определенные пользователем

В добавление к ниже изучаемым классам, типы, определенные пользователем, включают перечисления, представляющие обычно небольшое множество фиксированных значений:

```
enum Week_day {Monday, Tuesday, Wednesday, Thursday, Friday, Saturday, Sunday};
enum Error {Division_by_zero, Null_pointer_dereference, File_error, Memory_error};
```

Внутренне значения перечисления являются целыми константами. Тип перечисления неявно отображается в целочисленный тип, но обратное неверно, поскольку значения переменной перечисления находится в определенных границах.

Еще одним видом типа, определяемого пользователем, является тип, задающий структуры, обычно называемый просто «struct» и представляющий некоторую форму класса. Классы изучаются ниже. У структур в основном те же свойства, что и у классов, но другая политика экспорта.

Наконец, тип «объединение» описывает объекты, которые могут быть нескольких видов, занимая одну и ту же область памяти. Вместо ключевых слов class или struct для этого типа применяется термин union. Этот механизм, перешедший от C, где он был предназначен для оптимизации памяти, не является типо-безопасным, так как при использовании p.a нельзя гарантировать, что p является правильным вариантом, имеющим атрибут a. По этой причине программисты C++ редко используют тип объединения, основываясь вместо этого на наследовании, как в других ОО-языках, для поддержки вариантов общего типа.

Классы

Класс C++ задается следующим синтаксисом:

```
class A{
    …// список членов класса
};
```

Определение может появиться в любом месте модуля трансляции, где допускаются определения, включая появление в объявлении другого класса и даже внутри тела функции, хотя такая вложенность — нечто экзотическое. Предпочтительно, но не требуется, включать определение класса в отдельный файл, как это вы бы сделали в Eiffel.

Класс можно использовать как тип при объявлении переменных и для конструирования других производных и определенных пользователем типов:

```
A var;                          // Переменная типа A
const A const_var;          // Константа типа A
A* p;                           // Указатель на объекты типа A
```

В C++ члены класса разделяются на члены-переменные, также называемые членами данных (соответствующие атрибутам Eiffel), и члены-функции, называемые также методами.

Определение класса может содержать как определения, так и неопределяющие объявления членов. В последнем случае эти члены должны быть определены вне класса.

```
class A{
       int n;                          // член-переменная
```

```
        void f ()                  // член-функция, определенная в классе
              {…}
        void g ();                 // член-функция, определенная вне класса
};
void A::g ()//Определение g
{…}
```

В C++ не применяется унифицированный принцип доступа — доступ к переменной и функции без аргументов рассматриваются как различные операции. Для функций всегда следует использовать круглые скобки, окаймляющие список аргументов, даже если он пуст:

```
A var;
int i;
i = var.n;
var.f ();
```

Еще одно важное отличие от Eiffel (и отклонение от принципа скрытия информации) состоит в предоставлении клиентам возможности непосредственного присваивания значений полям класса:

```
var.n = 5;
```

Нужно ли говорить, что это не рекомендуется, — используйте сеттер-процедуры для этого.

Специальные члены функции, называемые конструкторами, служат для создания объектов. Вот пример конструктора:

```
class A{
    int n;
                 // Следующие две строки определяют конструктор
    A (int i)
       {n = i;}
…
};
```

Конструктор играет ту же роль, что и процедура создания в Eiffel, но есть важные различия. Как показывает пример, конструкторы не имеют своих собственных имен, а используют имя класса, здесь – A. Появляющееся в связи с этим ограничение требует, чтобы конструкторы отличались по сигнатуре (перегрузка подробнее будет изучаться ниже). Для инициализации переменной, тип которой задан классом, необходим вызов конструктора:

```
A var = A(2);
A another_var(2);
```

Здесь вторая форма является сокращением первой. Класс может иметь конструктор по умолчанию без аргументов (вспомните аналог, процедуру создания *default_create* из класса *ANY* в Eiffel, позволяющую опускать явную инициализацию):

```
A var;        // Синоним для A var = A ();
```

Если не задать в классе ни одного конструктора, то автоматически будет добавлен конструктор по умолчанию, который всем данным класса даст значения по умолчанию. Если, однако, какой-либо член класса принадлежит типу, у которого нет конструктора по умолчанию, то необходимо явно задать, по меньшей мере, один конструктор.

Конструктор предполагает, что первым делом все члены-переменные получают значения по умолчанию. Эти значения определены в языке для всех встроенных типов, а для классов задаются конструктором по умолчанию, если он присутствует. В его отсутствие необходимо обеспечить значение по умолчанию через список, инициализирующий члены, который может также включать любые другие члены данных, как в примере:

```
   class B {
       int value;
       Person a_friend;
       B (int v, string name) : value (v), a_friend (name) {}
              // Может также быть записано в виде:B (int v, string name) : a_friend
(name)
              // {value = v;}
   };
```

Процесс конструирования объекта включает конструирование полей, используя список инициализации, если он поставляется, или конструктор по умолчанию. Порядок инициализации определяется порядком определения переменных в классе. После этого выполняется тело конструктора.

Для обозначения текущего объекта(**Current** в Eiffel) используется ключевое слово this, определенное как константный указатель.

Для практики C++ характерно определение функций с побочным эффектом, изменяющих состояние и возвращающих значение, с нарушением тем самым принципа разделения команд и запросов. Ничто кроме привычки не мешает проектировщикам соблюдать этот принцип. Гарантировать, что функция не изменяет состояние, можно, если задать для нее модификатор const:

```
   class A {
       int n;
       int n_squared () const
       {return n_n; }
   };
```

Такие функции не могут модифицировать текущий объект, присваивая значения членам переменным или вызывая неконстантные функции

Скрытие информации

В Eiffel привилегии доступа клиентов определяются через спецификации экспорта, которые могут быть выборочными. В C++ можно определить для каждого члена класса один из трех модификаторов доступа: public, protected и private. Public-доступ означает полные права для клиентов и потомков (как с экспортом для *ANY* in Eiffel). Модификатор protected разрешает доступ потомкам. Модификатор private запрещает доступ и для потомков, оставляя доступ только для самого класса (в Eiffel потомки всегда имеют доступ для неквалифицированного вызова). Эти правила на самом деле одинаковы для квалифицированных и неквалифициро-

ванных вызовов (так что здесь нет эквивалента, позволяющего сделать компонент полностью недоступным для квалифицированного вызова, экспортируя его *NONE*). Следующие примеры используют некоторые модификаторы доступа:

```
class A {
private:
    int n;
    float secret_function () { … }
protected:
    float variable_for_descendants;
    int n_squared () { … }
public:
    string variable_for_everyone;
    do_everything () { … }
};
```

На практике члены-переменные редко задаются с модификатором public, так как это делает их доступными для присваивания, а не только для чтения. Общая практика состоит в определении геттеров, мы видели, почему в этом нет необходимости в Eiffel.

Модификатором по умолчанию для членов класса является модификатор private. Для структур, однако, таковым является public.

Эти приемы не поддерживают в полной мере понятие «выборочного экспорта» (**feature** {*A*, *B*, *C*} как в Eiffel, который позволяет экспортировать компонент классам, перечисленным в списке, и их потомкам). В C++ имеется, однако, близкий механизм «друзей» — класс может указать функцию или другой класс в качестве друга — friend:

```
class Linkable {
    friend class Linked_list;
    friend bool is_equal (const Linkable& other);
    …
};
```

Другу доступны все члены класса, включая private и protected (это означает, что уровень гранулярности механизма друзей грубее, чем выборочный экспорт, который позволяет предоставлять доступ к отдельным компонентам класса). В отличие от члена функции дружественная функция не вызывается на текущем объекте, так как она не может использовать this и может получать доступ только квалифицированным путем.

Область действия

Любая переменная, константа, функция или тип имеет область действия: локальную в блоке (часть программы, заключенная в фигурные скобки{...}), если там появилось ее объявление, в противном случае – глобальную (расширенную на весь модуль трансляции).

Как следствие, нет специального синтаксиса для локальных переменных, которые просто определяются где-либо в блоке, создаваемом в теле функции. Считается хорошей практикой для C++ определять локальные переменные как можно ближе к точке их первого использования.

Блоки могут быть вложенными. Объявление внутреннего блока скрывает элемент, определенный с тем же именем во внешнем блоке или глобальный:

```
int x;       // глобальная переменная x
void f ()
    {x = 1;      // Присваивание глобальной x
    int x;          // Определение локальной переменной x
    x = 2;      // Присваивание локальной x
}
int y = x;      // используется глобальная x
```

Скрытый элемент может быть все же доступен во внутреннем блоке, используя ::-нотацию – операцию разрешения области:

```
int x;                      // Глобальный x
class A {
    int x;                  // Член x
    void f ()
       {int x;              // Локальный x
       int y = A::x;        // Использование члена x
       int z = ::x;         // Использование глобального x
       }
};
```

Эти свойства чреваты ошибками — предпочтительнее для переменных вложенных областей выбирать различные имена.

Операции

Как в Eiffel, можно определять функции, вызываемые в стиле синтаксиса операций:

```
class Complex {
    …
    Complex operator+ (const Complex& other) const {…}
    Complex operator- (const Complex& other) const {…}
    Complex operator* (const Complex& other) const {…}
    Complex operator/ (const Complex& other) const {…}
};
```

В отличие от Eiffel, операции не имеют эквивалентного имени, заданного идентификатором, и ограничены фиксированным множеством предопределённых операций (таких как + и другие знаки), чьи синтаксические свойства – число аргументов, инфиксные или префиксные, лево- или правоассоциативные, приоритет – не могут быть изменены.

Перегрузка

В C++ разрешается в одной и той же области объявлять несколько функций с одним и тем же именем, если они отличаются по сигнатуре:

```
void print (int n) {…}
void print (string s) {…}
…
```

```
print (5);              // Использует функцию print печати целых
print ("hello");        // Использует функцию print печати строк
```

Этот механизм называется перегрузкой функций. Он часто используется, например, для операций (перегрузка операций). Перегруженными являются и конструкторы класса.

Статические объявления

В чистом ОО-каркасе все элементы программы являются относительными (по отношению к текущему объекту, мы называем это «общей относительностью»). В C++ добавляется механизм, описывающий члены-переменные и функции как статические. Такие члены принадлежат классу, но могут применяться независимо от его экземпляров (в Eiffel подобный эффект имеет место для однократных — **once**-методов, не используемых в этой книге, но описанных в стандарте языка).

В отличие от обычных членов переменных, которые представляют поля каждого экземпляра класса, *статические члены-переменные* представляют данные уровня самого класса. Например, статический член данных может применяться для подсчета числа вызовов некоторой функции класса:

```
class Rocket_launcher {
    …
    static int rocket_count;
    static const int max_count = 100;
    void launch ()           // Запуск ракеты
    {
       …
       rocket_count++;     // Увеличение на 1 при очередном запуске
    }
};
```

Статические члены функции оперируют только со статическими членами переменных и константами, как в примере[1]:

```
class Rocket_launcher {
    … Rest of class as above …
    static bool is_in_bounds ()
    {return rocket_count <= max_count;}
};
```

Вызовы статических членов не требуют целевого объекта. Вместо точечной нотации используется операция разрешения области:

```
r = Rocket_launcher::rocket_count;
m = Rocket_launcher::max_count;
if (Rocket_launcher::is_in_bounds ()) …
```

[1] В статических функциях разрешается вызов других статических функций. При обсуждении статических классов и компонентов в Java и в C# я уже высказывал свою точку зрения, что эта концепция согласуется с ОО-подходом – есть статический конструктор, который автоматически создает статический объект, доступный клиентам и экземплярам самого класса.

Совсем другое по семантике использование ключевого слова static применяется при описании локальных переменных функции. Если локальная переменная объявляется как статическая, то сохраняется ее значение, полученное при предыдущем вызове функции, как показано в данном примере:

```
void f ()
       { static int invocation_count = 0;     // локальная статическая
       …
       invocation_count++;                  // Увеличивается на 1 при каждом вызове
}
```

Время жизни объектов

В C++ есть дифференциация объектов на тех, что автоматически управляются системой, и тех, что находятся под прямым контролем программиста.

Объекты, управляемые программистом, называются также динамическими объектами, хранимыми в области памяти, которая называется «куча». Программисты создают объекты, вызывая операцию **new** с соответствующим конструктором класса:

```
Complex* c;
…
c = new Complex (1.0, 2.0);
```

Вычисление выражения new создает указатель на объект типа Complex, который и присваивается соответствующей переменной, — эффект тот же, что и в Eiffel при вызове процедуры создания **create** *c.make* (1.0, 2.0) с типом *COMPLEX*.

В отличие от Eiffel, C++ не проектировался для автоматической сборки мусора. В его распоряжении есть оператор удаления объектов из кучи – delete:

```
delete c;
```

Для тех же самых целей C обеспечивает библиотечную функцию, называемую free. Если после выполнения этой операции некоторые указатели или ссылки будут все еще ссылаться на объект, который до этого был присоединен к *c*, то они станут висячими указателями: разыменование такого указателя означает попытку доступа к несуществующему объекту и приведет к ошибке периода выполнения или, что еще хуже, выработке произвольного значения. Но, с другой стороны, если не удалять недостижимые объекты, это может стать причиной исчерпания памяти.

В то время как динамические объекты всегда доступны через указатели – с возможностью нескольких указателей быть присоединенными к одному и тому же объекту, — нединамические объекты (которые все же могут иметь «вторичные» указатели и ссылки, присоединенные к ним) связываются с единственной переменной или константой, чья область и определяет время жизни объекта. Нединамические объекты могут быть двух видов.

- Локальные переменные функций или других блоков, для которых объекты создаются автоматически (в стеке). Они управляются стеком вызовов, распределяются в точке определения и забываются при выходе из блока.
- Глобальные переменные и переменные, объявленные как статические (статические данные класса, статические локальные переменные функции), они создаются до начала выполнения функции запуска — main и существуют до конца выполнения системы.

Время жизни полей объекта (соответствующих нестатическим членам-переменным класса) определяется временем жизни самого объекта. Процесс разрушения начинается с самого объекта, а затем разрушаются его поля. В отсутствие автоматической сборки мусора часто необходимо специфицировать некоторые операции, выполняемые при удалении объекта (либо при явном вызове delete, либо при выходе из блока). Любой класс Т в С++ может определить для этих целей специальный член-функцию – деструктор, со стандартно построенным именем ~Т, который будет вызываться при разрушении объекта. Вот пример типичного деструктора:

```
class Person1 {
…
    Passport* pp;
    Person1 (string n, date d)// Конструктор, создающий объект Passport
       {pp = new Passport (n, d); }
    ~Person1 ()             // Деструктор
       {delete pp;}    // Удаление объекта passport
};
```

Этот пример иллюстрирует общий С++ образец: инициализация как способ овладения ресурсом (Resource Acquisition Is Initialization — RAII) – использование конструктора для захвата ресурсов, необходимых объекту, и деструктора, для их освобождения. Это устраняет некоторые источники ошибок при уверенности, что операции удаления – delete – выполняются в правильном порядке. Систематическое применение RAII ограничивает использование динамической памяти специальными классами, часто библиотечными. В остальной части ПО экземпляры этих классов используются не динамически, ослабляя тем самым последствия отсутствия сборки мусора.

Образец RAII расширяется на ресурсы, иные, чем память, такие как файлы, сокеты (сетевые соединения) и блокировки (для многопоточности и других форм параллельного программирования), для которых освобождение ресурсов должно выполняться вручную, даже при наличии сборки мусора. Кроме того, RAII гарантирует, что в случае ненормального завершения (через исключения, обсуждаемые ниже) деструкторы будут вызваны подходящим образом. Эти преимущества приводят некоторые защитники, полагая, что RAII превосходит сборку мусора. Однако он остается ручным подходом, ограниченным специфическими образцами использования памяти.

Инициализация

В отличие от Eiffel, автоматическая инициализация применима к статическим объектам (предустановленными значениями для встроенных типов, конструктор по умолчанию для типов, заданных классами). Вернемся к объявлению:

```
int n;
class Rocket_launcher {
    static int rocket_count;
    … Остальное как выше …
};
int Rocket_launcher::rocket_count;
```

Здесь как n, так и rocket_count инициализируются нулем (заметьте: необходимо включать второе объявление rocket_count, так как объявление статического члена переменной внутри класса не является определяющим объявлением).

Ссылки, константы и автоматические объекты необходимо инициализировать вручную.

Отсутствие инициализации автоматических объектов приводит к тому, что значение будет неопределенным, а это почти всегда становится причиной ошибки (и источником потенциальных нарушений безопасности). Так что следует тщательно проверять вручную, чтобы каждый автоматический объект имел подходящую инициализацию.

Обработка исключений

В С++ обработка исключений доступна через процесс исключительных событий, возникающих в период выполнения. Вместо стиля Eiffel, основанного на принципе проектирования по контракту, используется стиль «try-catch».

Исключение, причиной которого, например, явилась ошибочная арифметическая операция (переполнение сверху или снизу — overflow или underflow), прервет нормальный поток выполнения. Блок кода со специальным синтаксисом, в котором контролируется возникновение исключительных ситуаций, называется охраняемым блоком – try-блоком. Исключительная ситуация, возникшая в охраняемом блоке, может быть перехвачена и обработана в одном из блоков обработки – catch-блоке. Рассмотрим пример:

```
try {
    … Код, который может включить исключение …
} catch (io_error e) {
    … Обработка исключений ввода-вывода I/O …
} catch (memory_error e) {
    … Обработка исключений, связанных с памятью …
}
```

Блок catch задает тип исключения, такой как io_error, и имя объекта, задающего исключение, здесь *e*, используемое в операторах обработки (способом, подобным формальным аргументам метода) для получения доступа к специфическим свойствам исключения.

Исключения можно также включать (говорят также – «выбрасывать»), применяя специальный оператор:

```
throw exp
```

Здесь *exp* является выражением. Хотя оно может быть любого типа, но на практике используются специальные библиотечные классы, спроектированные для описания исключений.

Любое исключение, встретившееся во время выполнения блока, прерывает выполнение этого блока (оставшиеся операторы не выполняются). После чего:

- если блок был охраняемым и один из catch-блоков соответствует типу возникшего исключения, то выполнение будет передано в соответствующий catch-блок, затем будет выполняться следующая конструкция, если только сам catch-блок не выбросит повторно исключение — throw ();
- если нет соответствующего обработчика или исключение возникло вне охраняемого try-блока, текущая функция завершается, выбрасывая исключение в вызывателе – функции, вызвавшей функцию, в которой возникло исключение. И здесь рекурсивно применяется описанная схема.

Если в цепочке вызовов не найдется соответствующий catch-блок, то выполнение закончится прерыванием начальной точки – функции main и, следовательно, завершением программы в состоянии, свидетельствующем об ошибке.

В этом процессе завершения функций и передачи исключения вверх по цепочке вызовов, известном как раскрутка стека, все автоматические объекты разрушаются, используя деструкторы, если они доступны. Это одно из тех мест, где помогает RAII.

Для безопасной обработки исключений можно указать, как часть сигнатуры функции, множество throw – список исключений, которые могут быть выброшены при выполнении. Вот пример:

```
void read_and_store (string a_file_name) throw (Io_error, Memory_error)
{ … }
```

Отсутствие множества throw может означать, что функция может выбрасывать любое исключение (для указания, что она не может выбрасывать исключения, применяется конструкция throw ()). Систематическое включение множества throw в каждую функцию означает, что для данной функции это множество является надмножеством множеств throw для всех вызываемых функций. Это рекомендуемая дисциплина, помогающая избежать пропуски исключений. Но это трудная задача, поскольку библиотеки и существующий код, используемый новыми системами, может не следовать этой дисциплине.

Шаблоны

Шаблоны являются С++-версией универсальности. Вот простой пример:

```
template <typename G> class Stack {
…
public:
    G item () {…}
    void push (G an_item) {…}
    void pop () {…}
};
```

Здесь определен класс Stack с родовым параметром G, аналог класса *STACK* [*G*] в Eiffel. Конкретный стек задается как:

```
Stack<int> s;
```

Главная разница между шаблонами С++ и универсальными классами, представленными в других ОО-языках, в том, что каждое родовое порождение, такое как выше, рассматривается как создание нового класса, — процесс, называемый конкретизацией шаблона.

> Препроцессирование – преобразование текста программы до компиляции, используемое в данном случае, сохраняет мощь языка программирования и дает некоторые экзотические применения в продвинутом С++-программировании.

Здесь нет аналога понятия ограниченной универсальности: если применяется операция к переменной, тип которой задан формальным параметром, таким как G из примера выше,

проверка типа будет применяться к каждому экземпляру, чтобы убедиться, что операция всегда правильна.

Шаблон класса позволяет задать полную или частичную специализацию. Полная специализация замораживает фактические параметры, как в этом примере, используя вышеприведенный Stack:

```
template<>
class Stack<bool> {
        … Операции, специфические для булевских стеков …
};
```

Частичная спецификация оставляет формальные параметры, задавая некоторые ограничения:

```
template<typename G>
class Stack<G*> {
        … Операции, специфические для стеков указателей …
}
```

Как показывают примеры, класс, полученный в результате специализации, может иметь собственные определения членов. Это не становится причиной конфликтов, так как компиляция всегда выбирает наиболее специализированную конкретную версию.

Помимо шаблонов классов, C++ поддерживает шаблоны функций:

```
template <typename G>
G max (G a, G b) { … Вычисление максимума …}
```

При конкретизации функции можно опускать фактические параметры шаблона, если их возможно автоматически вывести из фактических типов аргументов, как в следующем примере:

```
int a, b, c;
…
c = max (a, b);     // Вызов max<int>
```

Здесь автоматически порождается max с типом int для G.

Механизм шаблонов выходит за пределы универсальности, позволяя в качестве параметров шаблона задавать конкретные типы, такие как булевские или целые типы. Соответствующие фактические родовые параметры должны быть константами периода компиляции. В следующем примере используется эта возможность для определения умножения матриц с гарантией соответствия размеров:

```
template <int n, int m> class Matrix { … };
template <int n, int m, int k> Matrix<n, k> operator*
        (const Matrix<n, m>& m1, const Matrix<m, k>& m2)
{… Алгоритм умножения матриц … }
```

Попытка умножить матрицы несовместимых (константы) размеров приведет к ошибке, обнаруживаемой на этапе компиляции.

С.4. Наследование

Класс В можно определить как наследника (производный класс) класса A (базового класса для B):

```
class B : A {
      …
};
```

Переопределение

Здесь нет эквивалента переименования и отмены определения, как в Eiffel. Для переопределения наследуемой функции просто включите новое определение. Но следует быть внимательным — нужно сохранить исходную сигнатуру, так как в противном случае речь будет идти о перегрузке, и нет простого способа обнаружения таких ошибок.

Статус экспорта и наследование

По умолчанию наследуемые члены закрыты (private). Для изменения их статуса можно задать модификатор доступа — private, public или protected для отношения наследования, как в следующем примере:

```
class B : public A {…};
```

При такой спецификации каждый наследуемый элемент получает статус доступа, представляющий минимум из двух статусов – оригинального и статуса наследования. В данном примере, где статус наследования самый высокий, все члены A сохранят свой статус доступа в классе B.

Доступ в стиле Precursor

Для получения доступа к оригинальной версии переопределенной функции – эквивалент Precursor в Eiffel – можно использовать операцию разрешения области, если только эта версия не является private:

```
class B : public A {
    void b_function () // Не обязательно, чтобы это было переопределение r
    {
        A::r (); // Вызов метода из класса A
        …
    }};
```

Статическое и динамическое связывание

Мы видели, что динамическое связывание является главным вкладом объектной технологии в архитектуру построения ПО. Основное различие между С++ и ОО-языками, такими как Eiffel, в том, что связывание здесь по умолчанию статическое. Критерием, определяющим, какая версия функции будет вызываться для целевого объекта, является тип по объявлению, а не динамический тип объекта, присоединенного в момент выполнения. Чтобы для функции выполнялось динамическое связывание, необходимо объявить ее виртуальной:

```
class Rectangle {
…
    virtual void draw() {…} // Следует объявить как виртуальную
    virtual void rotate() {…}
};
class Rounded_rectangle : public Rectangle {
…
    virtual void draw() {…}      // Можно, но не обязательно объявлять виртуальной
                                 //при переопределении
    void rotate() {…}            // … Эффект тот же - динамическое связывание
                                 // будет применяться!)
};
```

Динамическое связывание применимо только для объектов, доступных через указатели или ссылки, как показано в следующем примере:

```
Rectangle r (1.2, 0.5);
Rounded_rectangle rr (5.0, 3.2, 0.2);
r = rr;          // r все еще обычный прямоугольник с полями, скопированными из rr
r.draw ();    // Статическое связывание …: rectangle::draw()
Rectangle*p = &rr;
p -> draw ();          // Динамическое связывание…: rounded_rectangle::draw()
Rectangle& ref = rr;
ref.draw ();           // Динамическое связывание…): rounded_rectangle::draw()
```

Конструкторы, которые не могут применяться к существующему целевому объекту, не могут быть виртуальными. Деструкторы могут, а часто и должны быть виртуальными, чтобы освобождать ресурсы, занятые данным динамическим объектом.

Чистые виртуальные функции

Ближайшим эквивалентом отложенного метода является понятие чистой виртуальной функции, имеющей определение, но не реализацию:

```
class Figure {
    virtual void draw() = 0;
    … Другие члены класса …
};
```

Класс, у которого есть хотя бы одна чистая виртуальная функция, называется абстрактным, он подобен отложенному классу

Множественное наследование

В C++ поддерживается множественное наследование:

```
class Arrayed_stack : public Stack, private Array {…}
```

Механизм менее гибок, чем представленный в этой книге. В частности, не поддерживается переименование. Можно наследовать две функции с одним именем и устранять конфликт, используя операцию разрешения области:

```
class A {void f () {…}};
class B {void f () {…}};
class C : public A, public B {…};
void test()
{
C* p = new C();
    // p->f();      Этот вызов неоднозначен и, следовательно, неправилен
    p->A::f();
    p->B::f();
}
```

Повторное наследование не позволяет выбирать между склеиванием и репликацией для каждого члена. Выбор делается глобально для класса в целом. Разрешение противоречий может требовать сложного использования области разрешения:

```
class D {int n;};
class E : public D{};
class F : public D {};
class G : public E, public F {};
void f()
{
    G* p = new G();
    // p->n = 0;                    // Это было бы неверным - какое n?
    p->E::D::n = 0;             // Правильно: присваивается версия из E
    p->F::D::n = 0;             // Правильно: присваивается версия из F
}
```

Если требуется, чтобы были присоединены общие поля предка вместо реплицированных, следует определить этого предка как виртуальный базовый класс:

```
class T {int n;};
class U : public virtual T {};
class V : public virtual T {};
class W : public U, public V {};
void f()
{
    W*p = new W;
    p->n = 0; // Теперь правильно
}
```

Неудобство в том, что выбор делается не в точке использования дублирующего наследования, здесь, в классе W, но ранее, в классах U и V, которые могут ничего не знать о планах W наследовать от них обоих.

Наследование и создание объекта

В C++ специфические правила создания экземпляров производных классов. Конструкторы не наследуются, но создание экземпляра производного класса становится причиной вызова конструктора родителя перед началом работы собственного конструктора (процесс рекурсивный). При вызове конструктора родителя ему могут быть переданы необходимые ему аргументы, как в следующем примере:

```
class Rounded_rectangle : public Rectangle {
public:
Rounded_rectangle (float w, float h, float r) : Rectangle (w, h), radius (r)
       {…}
… Остальная часть класса как ранее …
};
```

C.5. Дополнительные механизмы структурирования программ

В больших программных системах трудно избежать появления конфликта имен: система может использовать библиотеки, включающие классы с одинаковыми именами. C++ предоставляет пространства имен, или именованные блоки, единственная цель которых – ограничение области имен, объявленных в каждом блоке:

```
       // "some_library.h":
namespace some_library {
    class Parser {…};
    class Lexer {…}
    …
}
       // "your_program.cpp":
#include "some_library.h"
namespace your_program {
    class Parser {…}; // Неопределенности нет: области разные
}
```

Для разрешения любой неоднозначности достаточно использовать операцию разрешения области. Если это становится утомительным при частом использовании имени из другого пространства, можно ввести локальное имя, применяя using-нотацию:

```
using some_library::Lexer;
Lexer lexer;           // Сокращение для some_library::Lexer lexer
```

Это можно сделать глобально для всего пространства имен

```
using namespace some_library;
```

С.6. Отсутствующие элементы

Приведем краткий обзор механизмов, важных при изучении программирования, как они представлены в этой книге, но для них нет прямых эквивалентов в С++. Приведем некоторые соображения по поводу возможной эмуляции этих механизмов.

Контракты

В С++ не поддерживаются механизмы проектирования по контракту (предусловия, постусловия, инварианты класса и цикла, варианты цикла), играющие важную роль в современной методологии программирования, на разработке которой основана данная книга.

Некоторым утешением является то, что в С++ разрешается использовать оператор утверждения – assert:

```
assert b;
```

Здесь утверждается, что булевское выражение b должно иметь место в этой точке программы при каждом ее выполнении. При задании соответствующей опции компиляции, включающей проверку утверждений, если утверждение выполняется, то программа продолжает нормально выполняться, если же нет, то – удивительный результат – выдается сообщение и программа завершается (естественно ожидать возникновения соответствующей исключительной ситуации, которая могла быть перехвачена и обработана).

Этот механизм делает возможным использовать утверждения для отладки, но, конечно, недостаточен для всех других применений контрактов для спецификации классов и методов, документирования, проектирования и так далее.

Многие люди предложили расширения С++ или пакеты макросов (макрос – набор операторов препроцессора – своего рода процедура) для эмуляции проектирования по контракту. Поиск в Web с запросом «Design by Contract in C++» даст ссылки на многие из этих инструментальных средств, чье использование остается ограниченным, так как они не интегрированы в язык.

Агенты

Как отмечалось, С++ не имеет механизма агентов. Простой эффект вызова переменной, задающей функцию, может быть достигнут (как мы видели) через указатели функций. Более сложное решение использует понятие функтора, реализованного перегрузкой операции «()» – операции вызова функции. Объект «функтор» можно вызывать подобно любой другой функции. Неудобство в том, что требуется определить отдельный класс для каждого возможного числа формальных аргументов.

Ограниченная универсальность

Мы видели, что в С++ нет прямого соответствия ограниченной универсальности и что каждая конкретизация шаблона осуществляет собственную проверку типов. Это означает невозможность информировать клиентов, что родовой параметр представляет потомка определенного типа. Они узнают об этом, только нарушив это требование, когда конкретизация шаблона не будет компилироваться.

Методологическое правило – ограничение универсальности необходимо задавать неформально через комментарии в определении шаблона:

```
template <typename G> /* G должен быть потомком Comparable */
G max (G a, G b) { … }
```

Общая структура наследования

В C++ нет эквивалента класса, представляющего вершину в иерархии наследования, такого, как *ANY* в Eiffel, нет и класса – аналога *NONE*.

С.7. Специфические свойства языка

Мы уже встречались с несколькими свойствами C++, не доступными в Eiffel. Сейчас рассмотрим две другие особенности: аргументы по умолчанию и вложенные классы.

Аргументы по умолчанию

Для формального аргумента можно задать значение по умолчанию, позволяя при вызове опускать задание соответствующего аргумента. Если не все формальные аргументы имеют значения по умолчанию, то аргументы, имеющие значения по умолчанию, задаются после них:

```
void f (float x, float y, int n = 1, char c = !) { … }
f (1.2, 5.0, 2, 'a');         //можно задать значения всех фактических аргументов
f (1.2, 5.0, 2);              //c имеет значение по умолчанию '!'
f (1.2, 5.0);                 //n и c имеют значения по умолчанию 1, '!'
```

Можно также определить функцию с переменным числом аргументов, используя многоточие вместо списка формальных аргументов. Этот механизм не является безопасным по типу и более подходит для программирования на уровне C, чем для приложений, написанных для C++.

Вложенные классы

В C++, как отмечалось, допускаются многие формы вложенности. В частности, класс может быть объявлен внутри другого класса и даже функции. Вложенный класс называется членом того класса, в котором он объявлен и может иметь такой же статус доступа, как и другие члены класса, — private, protected или public.

Классы члены со статусом private представляют абстракции данных, спроектированные в интересах только охватывающего класса. Альтернативным решением является проектирование независимого класса и применение механизма друзей, но в этом случае все члены класса доступны «другу», в то время как вложенный класс может иметь закрытые члены, недоступные охватывающему классу.

С.8. Библиотеки

Часто используемая в приложениях C++ библиотека STL (Standard Template Library) покрывает фундаментальные структуры данных, в частности, контейнеры, требующие универсальности, так что большинство ее классов представляют шаблоны (отсюда название). В этой же библиотеке размещаются классы ввода-вывода и исключений.

Для ввода-вывода STL использует потоки, которые могут представлять окружение, такое как консоль (стандартные потоки *cin* и *cout*), файлы и строки. При чтении и записи применяются перегруженные операции побитового сдвига >> и <<, так что типичное взаимодействие с консолью выглядит примерно так:

```
Person p (…);
int my_age;
cout << "Name: " << p.name << endl << "Age: " << p.age () << endl;
cout << "Enter your age: " << endl;
cin >> my_age;
```

Здесь endl устанавливает конец строки.

Доступны и библиотеки третьих компаний.

В С++ сохраняются стандартные библиотеки С, которые предпочтительнее избегать, поскольку многие из их свойств низкоуровневые и небезопасны по типу.

С.9. Синтаксические и лексические аспекты

Грамматика С++ для операторов и выражений сложна; только базисные ее элементы будут рассмотрены.

Операторы как выражения

Ключевым понятием является операторное выражение – выражение, заканчивающееся точкой с запятой. Это понятие кажется парадоксальным, так как в этой книге проводится четкое разделение между операторами и выражениями, в соответствии с различием команд и запросов. В С++, однако, не настаивают на таком разделении, так что операторное выражение является как оператором, так и выражением, возвращающим значение, если только тип его отличен от void.

Соответственно, функция, возвращающая значение, может иметь побочный эффект. В С++ общепринято вызывать такую функцию, как оператор. В этом случае теряется возвращаемый результат.

Одним из следствий смешения концепций является то, что присваивание рассматривается как выражение, чье значение присваивается цели (в качестве побочного эффекта). Это делает возможным такие комбинации, как:

```
a = b = 5;
```

Здесь выражение справа равно 5, его значение присваивается b, результат присваивания, по-прежнему 5, присваивается a. Таких схем лучше избегать.

Управляющие структуры

Блоки соответствуют составному оператору Eiffel и состоят из списка операторов в фигурных скобках. Блоки, как мы видели, могут быть вложенными.

Условный оператор имеет форму:

```
if (expression) statement else statement
```

Заметьте, выражение условия – *expression* – должно быть в круглых скобках. Оно не обязано быть булевского типа, а может быть числовым и даже указателем — 0 и Null интерпретируются как **false**, остальные значение эквивалентны **true**. Здесь и в других структурах оператор может быть блоком. Принято даже одиночный оператор заключать в фигурные скобки, чтобы облегчить возможные добавления в будущем.

Здесь нет эквивалента elseif, так что необходимо использовать вложенность, но из-за отсутствия ключевого слова *end* и структурных отступов визуально вложенность не ощущается:

```
if (expression) statement
else if (expression) statement
else if (expression) statement
…
else statement
Оператор выбора имеет форму:
switch (expression) {
case value: statement; break;
case value: statement; break;
…
default: statement
}
```

Здесь *expression* задается булевским или целочисленным выражением, а каждое *value* представляет вычислимую в период компиляции константу. Если значение выражения не совпадает ни с одной константой, то выполняется ветвь default, если она задана, в противном случае ничего не делается (в Eiffel в отсутствие ветви **else** в операторе **inspect** в подобной ситуации в период выполнения генерируется ошибка). Оператор switch не задает конструкцию с одним входом и одним выходом, а представляет многоцелевой goto. Для правильной структурированности следует четко следовать показанной схеме. Оператор break, завершающий каждую ветвь, позволяет избежать типичной ошибки для C++ и C, когда управление проваливается в другую ветвь.

В C++ возможны три вида циклов:

```
while (expression) statement
do statement while (expression);
for (init_statement ; expression ; advance_statement) body_statement
```

Во всех этих вариантах *expression* служит условием продолжения. Это отличается от соглашения для формы **from** ... **until** ... **loop** ... **end**, используемой в этой книге, где **until**-выражение используется как условие выхода. Для преобразования условия из одной формы в другую достаточно применить операцию отрицания.

Разница между первыми двумя формами цикла состоит в точке проверки условия продолжения – в начале цикла или в конце. В первом варианте тело цикла может ни разу не выполняться, во втором — гарантируется, что тело цикла будет выполнено, по крайней мере, один раз.

Цикл for — наиболее общий и наиболее часто используемый. Цель *advance_statement* — обеспечить продвижение к следующей операции (в Eiffel эта часть включается в тело цикла). Приведем пример цикла в Eiffel:

```
from i := 1 until i > n loop
    …
    i := i + 1
end
```

Его эквивалент в C++:

```
for (int i = 1; i <= n; i++)
        {…}
```

В C++ используются goto-подобные операторы: сам goto, применять который не рекомендуется, оператор break, появившийся в связи со switch, и return, применяемый для возврата значения функцией:

```
return expression;
```

Оператор завершает выполнение и возвращает заданное значение (для процедур C++ – функций, возвращающих void – выражение *expression* опускается).

Как результат, блоки C++ не ограничиваются структурой с одним входом и одним выходом, которая систематически используется в этой книге в соответствии с рекомендациями методологии программирования.

Присваивание и его расширения

Цель присваивания не обязана быть переменной — она должна обозначать область в памяти (называемую «left-value» или «l-value», так как появляется слева от символа присваивания). Вот несколько примеров:

```
int a; Person p;
a = 5;               // Правильно
// a + 2 = 5;              Возникла бы ошибка, поскольку a +2 не связана с областью
                         // памяти
*(&a + 1) = 5;           // Правильно: присваивание области памяти, следующей за a
p.name = "Izzy";         // Правильно: присваивание области памяти, отведенной полю
                         // name объекта p
```

Некоторые C++ операции вместе с присваиванием выполняют заданную операцию. В частности:

- a += b – это краткая запись для a = a + b, аналогичный смысл и для других операций, отличных от +;
- для выражения a = a + 1 существует еще более краткая форма: a++ или ++a. Как обычно, это выражения, которые могут играть роль операторов. Разница в том, что первое выражение возвращает в качестве результата a, второе – увеличенное на 1 значение. Потом уже возникает побочный эффект (тест: каков эффект присваиваний a = a++ и a = ++a?).

Использование для присваивания знака равенства, а для эквивалентности двойного равенства – это отход от многовековой математической традиции. В сочетании со слабой ти-

пизацией булевских выражений, где истиной является все, что угодно, это приводит к типичной ошибке, встречающейся во многих программах на C++:

```
if (x = y) {Some_instructions}
```

На этом попадаются даже опытные программисты — скорее всего, требовалась проверка на равенство, а получилось присваивание. Компилятор все это пропустит и с большой вероятностью выполнит Some_instructuions, если только *y* отлично от нуля.

Выражения и операции

В C++ выражение является литералом, идентификатором, this или выражением со знаками операций. Следующая таблица включает все операции C++. Унарными операциями являются:

Операция	Роль	Пример	Операция	Роль	Пример
+	Унарный плюс	+a	Delete	Освобождение памяти	delete p
-	Унарный минус	–a	Sizeof	Размер типа выражения	sizeof (a + b)
*	Разыменование	*p	++	префиксное увеличение	++a
~	Побитовое отрицание	~a	++	постфиксное увеличение	a++
!	Логическое отрицание	!b	--	префиксное уменьшение	--a
new	Выделение памяти	new int (5)	--	постфиксное уменьшение	a--

Бинарными операциями являются:

<table>
<tr><th>Операция</th><th>Роль</th><th>Пример</th><th>Операция</th><th>Роль</th><th>Пример</th></tr>
<tr><td>+</td><td>Бинарный плюс</td><td>a + b</td><td>=</td><td>Присваивание</td><td>a = 5</td></tr>
<tr><td>-</td><td>Бинарный минус</td><td>a - b</td><td>+=</td><td>Присваивание плюс</td><td>a += 5</td></tr>
<tr><td>*</td><td>Умножение</td><td>a * b</td><td>-=</td><td>Присваивание минус</td><td>a -= 5</td></tr>
<tr><td>/</td><td>Деление</td><td>a / b</td><td>*=</td><td>Присваивание умножить</td><td>a *= 5</td></tr>
<tr><td>%</td><td>Взятие по модулю</td><td>a % b</td><td>/=</td><td>Присваивание делить</td><td>a /= 5</td></tr>
<tr><td>^</td><td>Побитовое xor</td><td>a ^ b</td><td>%=</td><td>Присваивание по модулю</td><td>a %= 5</td></tr>
<tr><td>&</td><td>Побитовое and</td><td>a & b</td><td>^=</td><td>Присваивание xor</td><td>a ^= b</td></tr>
<tr><td>|</td><td>Побитовое or</td><td>a | b</td><td>&=</td><td>Присваивание and</td><td>a &= b</td></tr>
<tr><td>&&</td><td>Логическое and</td><td>b1 && b2</td><td>|=</td><td>Присваивание or</td><td>a |= b</td></tr>
<tr><td>‖</td><td>Логическое or</td><td>b1 ‖ b2</td><td><<=</td><td>Присваивание сдвиг влево</td><td>a <<= 1</td></tr>
<tr><td>==</td><td>Эквивалентно</td><td>a == 5</td><td>>>=</td><td>Присваивание сдвиг вправо</td><td>a >>= 1</td></tr>
<tr><td>!=</td><td>Не эквивалентно</td><td>a != 5</td><td>[...]</td><td>Взятие индекса</td><td>a [i]</td></tr>
<tr><td><</td><td>Меньше чем</td><td>a < 5</td><td>,</td><td>Последовательность</td><td>a, b = 2</td></tr>
<tr><td><=</td><td>Меньше или равно</td><td>a <= 5</td><td>.</td><td>Доступ к члену</td><td>x.f</td></tr>
<tr><td>></td><td>Больше чем</td><td>a > 5</td><td>._</td><td>Непрямой доступ к члену</td><td>x.*pf</td></tr>
<tr><td>>=</td><td>Больше или равно</td><td>a >= 5</td><td>–></td><td>Доступ через указатель</td><td>px–>f</td></tr>
<tr><td><<</td><td>Побитовый сдвиг влево</td><td>a << 1</td><td>–>*</td><td>Непрямой доступ через указатель</td><td>px–>*pf</td></tr>
<tr><td>>></td><td>Побитовый сдвиг вправо</td><td>a >> 1</td><td>::</td><td>Разрешение области</td><td>Person::name</td></tr>
</table>

Операция деления адаптирована к типам операндов: для целых операндов – это деление нацело; если хотя бы один операнд с плавающей точкой, то деление с плавающей точкой.
Операция «последовательность» (запятая) в духе языка – слияние операторов и выражений; последовательность выражений вычисляется слева направо, результат – значение последнего выражения. Пример демонстрирует возможность краткой записи свопинга – обмена значениями двух переменных – с помощью одного присваивания:

```
b = (temp = a, a = b, temp)
```

Нужно ли говорить, что предпочитать нужно ясную запись, даже если она длиннее.

В C++ поддерживается понятие условного выражения, задаваемого в форме:

```
x ? a : b
```

В этой тернарной операции x, a, b – выражения; x интерпретируется как булевское, если оно истинно, то результатом является значение a, в противном случае – b. Вот образец типичного использования:

```
template <typename G>
G max (G a, G b) { return a > b ? a : b; }
```

При вызове функции применяются круглые скобки, окаймляющие список аргументов. Скобки также рассматриваются как операция.

Программисты могут перегружать все операции, за исключением следующих четырех:

```
.      .*     ::      :?     sizeof
```

Идентификаторы

Идентификатором в C++ является любая последовательность букв и цифр, начинающаяся с буквы (подчеркивание относится к буквам, по соглашению, идентификаторы, начинающиеся с подчеркивания, резервируются для специальных переменных, управляющих компиляцией).

В отличие от Eiffel, идентификаторы C++ чувствительны к регистру.

Здесь нет стандартных соглашений по наименованию, так что можно использовать соглашения этой книги или другие правила стиля. Заметьте, однако, что в STL имена классов заданы в нижнем регистре.

Литералы

Литералы (манифестные константы) могут представлять целые, символы, строки и числа с плавающей точкой.

Целая константа может быть десятичной, восьмеричной, начинающейся цифрой 0, шестнадцатеричной с предшествующими символами 0x. Десятичное число 12 можно записать тремя константами:

```
12              // Десятичная
014             // Восьмеричная
0xC             // Шестнадцатеричная
```

Будьте внимательны, не начинайте нулем десятичные константы – константа 012 интерпретируется как восьмеричная, ее значение –10.

Символьные константы заключаются в одиночные кавычки – 'A'. Константы с плавающей точкой состоят из целой части, десятичной точки, дробной части и, возможно, целой экспоненты, состоящей из символа e, за которым следует целое, возможно со знаком. По умолчанию такая константа относится к типу double, если только она не заканчивается символом f, указывающим на тип float, или *l* – тогда тип long double. Строковая константа – это последовательность символов, заключенная в парные кавычки.

Ключевые слова

Следующие имена зарезервированы в C++ для использования в качестве ключевых слов:

```
asm, auto, break, case, catch, char, class, const, continue, default, delete, do,
double, else, enum,extern, float, for, friend, goto, if, inline, int, long, new,
operator, private, protected, public, register,return, short, signed, sizeof,
static, struct, switch, template, this, throw, try, typedef, union,unsigned,
virtual, void, volatile, while.
```

C.10. Дальнейшее чтение

Руководство от автора языка (всякий, кто серьезно интересуется C++ должен прочитать его):

Bjarne Stroustrup: The C++ programming language, 3rd edition, Addison-Wesley, 2000.

Последнее издание на русском языке: Бьёрн Страуструп, Язык программирования C++, Специальное издание, Бином, 2008 г.

Вводные тексты:

Herbert Schildt: C++: A Beginner's Guide, McGraw-Hill, 2003

На русском языке: Герберт Шилдт, Самоучитель по C++, 3-е издание, БХВ –Петербург, 2002 г.

Bruce Eckel: Thinking in C++: Introduction to Standard C++, Prentice Hall, 2000.

Для продвинутых свойств, особенно для программирования, основанного на шаблонах:

Andrei Alexandrescu: Modern C++ Design: Generic Programming and Design Patterns Applied, Addison-Wesley, 2001.

David Vandevoorde and Nicolai M. Josuttis: C++ Templates: The Complete Guide, Addison-Wesley, 2002.

David Abrahams and Aleksey Gurtovoy: C++ Template Metaprogramming: Concepts, Tools and Techniques from Boost and Beyond, Pearson, 2004.

D

От C++ к C

Язык C++ является расширением C и представляет редкий пример языка почти полной обратной совместимости с предшественником – правильная C-программа обычно является и правильной C++ программой и дает те же результаты.

Язык C++ не заменяет C. В своем исходном состоянии язык C сохранил свою значимость для хорошо определенной области приложений – программирование на уровне операционной системы и аппаратуры. В частности, поэтому едва ли любой процессор существует без C-компилятора. А может быть все наоборот: никто не отважится выпустить процессор без C-компилятора, поскольку этого ожидает рынок. В любом случае C де-факто является стандартом для низкоуровневого программирования.

Большинство C-программистов знают и C++. По этой причине в данном приложении не описываются основы языка: предполагается, что вы прочли предыдущее приложение, так что здесь просто перечисляются конструкции C++, недоступные в C. Это можно сделать быстро, так что приложение короткое.

Такая концепция описания (указание различий с другим языком) объясняет также, почему, в отличие от других приложений, обсуждение основ и стиля C вынесено в самый конец.

D.1. Отсутствующие элементы

Исходя из описания C++, нисходя от C++ к C, следует опустить:

- все ОО-механизмы: классы и объекты, члены функции (для структур), наследование, виртуальные функции, конструкторы, деструкторы, ссылки (но указатели остаются);
- шаблоны (в C нет универсальности);
- аргументы по умолчанию (функции с переменным числом аргументов доступны через библиотечный механизм, известный как varargs.);
- механизмы управления доступом и friend – механизм друзей, а также операцию разрешения области;
- пространства имен;
- исключения.

Большинство из того, что осталось, включает:

- статические функции;
- слияние операторов и выражений;
- указатели (не ссылки) и возможность манипулировать ими, используя адресную арифметику;
- структуры управления, кроме исключений;
- операции с побочным эффектом, такие как ++;

- синтаксические соглашения: скобки вместо ключевых слов, знак = для присваивания, точка с запятой для завершения операторов;
- доступность препроцессора, в частности, переменных времени компиляции, позволяющих управлять условной компиляцией (#ifdef compile_time_variable, где значение compile_time_variable устанавливается вне программы, например, как опция компиляции).

D.2. Основы языка и стиль

Язык С возник в результате исследований, проводимых в AT&T's Bell Laboratories в конце шестидесятых с целью получения преимуществ от идей структурного программирования, с сохранением при этом возможности прямого доступа к механизмам машинного уровня. Последнее требование связано не только с проблемами производительности; другая причина состояла в том, что на С предстояло написать операционную систему – первую версию Unix. Этот проект был успешным, и все последующие версии Unix (и нескольких других операционных систем) были написаны на С.

Язык С ценится за возможность управления низкоуровневыми аспектами приложения – то, что называется возможностью работы на уровне операционной системы и аппаратуры. Такая работа поддерживается возможностью прямого манипулирования адресами, в частности, использованием указателей (* и &-операций), адресной арифметики и идеи о том, что целью присваивания может быть любое выражение (l-value), которое может обозначать адрес.

Ничто не дается даром, так что, достигая хорошего контроля над ресурсами низкого уровня, теряем преимущества абстракции, обеспечиваемой более современными языками, в частности, такого аспекта, как проверка типов. Динамически разрешается определить любой адрес, но нет гарантии, что при каждом выполнении по этому адресу будет находиться предполагаемое по смыслу значение. Это не просто проблема надежности, но также и проблема безопасности. Переполнение буфера, один из любимых способов атаки интернетовских разбойников, фундаментально основан на С-механизме доступа к произвольному адресу памяти, вычисляемого динамически.

Компромисс устанавливает пределы разумного использования С. Хотя С продолжает применяться для разработки больших систем, но это не лучшее его использование. Две важных области применения остаются для С: небольшие программы для прямого доступа к ресурсам и целевой язык для переносимых компиляторов.

В своей первой роли С остается инструментом прямого использования программистами. Наблюдения показывают, что для основной части любого приложения нет нужды в низкоуровневых аспектах С, они будут только страдать от них, например, рискуя получить самые неприятные ошибки доступа к памяти в период выполнения. Некоторой специализированной части приложения может, однако, понадобиться прямое взаимодействие с платформой (аппаратура плюс операционная система). Программисты должны обеспечить эти механизмы в форме четко специфицированных и тщательно прописанных функций, типично написанных на С и точно так же типично коротких. Примером может быть процедура, посылающая информацию через сокет (абстрактное сетевое соединение). Такие программы, обычно не более нескольких строчек или нескольких десятков строчек, должны быть сгруппированы в библиотеку и доступны остальной части ПО через тщательно разработанный интерфейс – API.

Библиотека EiffelBase использует этот подход при реализации таких абстракций, как массивы и файлы. Для остального мира соответствующие классы являются нормальными Eiffel-классами с контрактами; их реализации просто вызывают короткие внешние С-функции.

В своей второй роли C служит целевым языком для компилятора некоторого языка программирования – ЯП, предлагающего более высокий уровень абстракции, чем C. Такой прием представляет важные преимущества.

- Почти универсальная доступность C-компиляторов облегчает конструирование переносимых компиляторов (где переносимость означает возможность поддержки разных платформ). Компилятор ЯП может сконцентрироваться на аспектах компиляции ЯП-программ, не зависящих от платформы, создавая результирующий код на C и оставляя C-компилятору задачу дальнейшего преобразования кода в машинный код для соответствующей платформы.
- Генерируемый C код может все же включать и элементы, зависящие от платформы, используя возможности условной компиляции.
- Технология C-компиляции хорошо понятна; значительная работа по оптимизации выполняется C-компиляторами. Авторы ЯП-компиляторов могут концентрироваться на ЯП-специфических оптимизациях и могут обычно полагаться на C-компилятор для выполнения стандартных оптимизаций конструкций нижнего уровня, таких как арифметические выражения, оптимизацию вычислений которых не нужно делать для каждого частного языка.

Этот подход успешно используется многими компиляторами, включая Eiffel-компиляторы.

Другие применения C, отличные от двух только что описанных, кажется, с трудом будут преодолевать ограничения C. Тем не менее, в своей области создано множество полезных решений за сорок лет существования этого высоко успешного языка.

D.3. Дальнейшее чтение

Brian W. Kernighan and Dennis M. Ritchie: The C Programming Language, second edition, Prentice Hall, 1988.

На русском языке это букинистическая редкость, но доступна на многих сайтах.

Библия C-программирования от авторов языка, известная как «K&R», ценимая за ясность и выразительность. Есть множество других книг по C, но трудно, кажется, прочесть что-либо другое, отличное от «K&R», для овладения C.

E

Использование среды EiffelStudio

На протяжении этой книги вам предлагалось писать и выполнять примеры в среде EiffelStudio. Данное приложение поможет вам подготовить примеры и запустить их на выполнение. Более точно – здесь вы найдете только основную справочную информацию; за деталями следует обратиться к расширенной версии, которая доступна онлайн на touch.ethz.ch/eiffelstudio.

E.1. Основы EiffelStudio

EiffelStudio – это общецелевая интегрированная среда разработки (Integrated Development Environment - IDE). Она является результатом многолетней разработки и повседневно при-

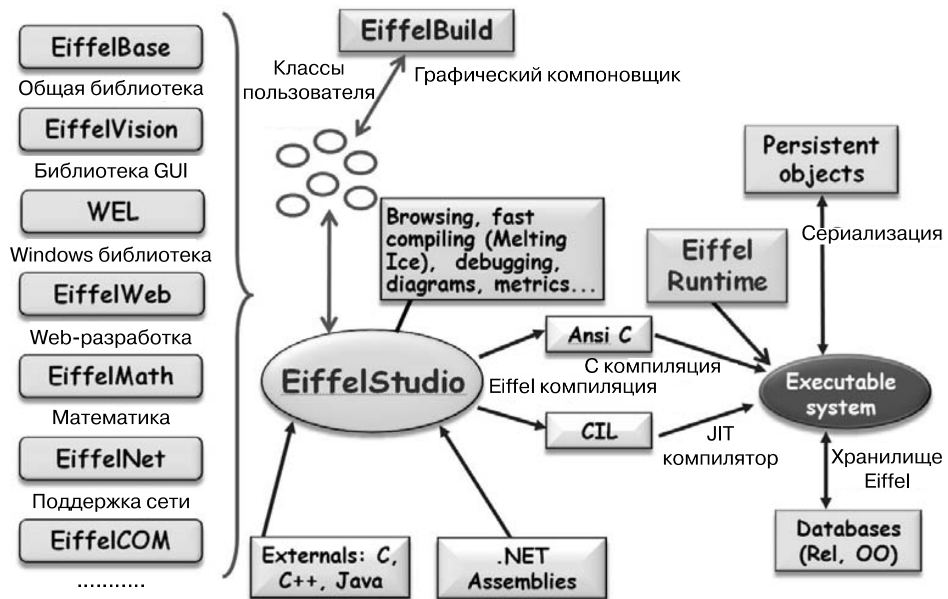


Рис. 12.18. главные компоненты EiffelStudio

меняется для создания программных систем, больших и малых, в различных областях приложения.

Кстати, одной из таких систем является сама EiffelStudio, написанная полностью на Eiffel, за исключением некоторого кода на С, реализующего систему исполнения («run-time system»). К моменту написания этого текста EiffelStudio включала примерно 2.2 миллиона строк кода, написанного в соответствии с принципами этой книги. Если вам интересно изучить этот код, не стесняйтесь – все окружение доступно как открытый код.

EiffelStudio имеет версии для различных операционных систем, включая Windows, Linux и другие варианты Unix. Детали инсталляции и запуска EiffelStudio варьируются на каждой платформе, и графическое представление («посмотри и почувствуй») соответствует стандартным соглашениям каждой поддерживаемой платформы, но среда в любом случае работает идентично.

Пояснения в этом приложении не зависят от платформы, за одним исключением – используется термин «каталог» (directory) вместо привычного для пользователей Windows термина «папка» (folder).

На рисунке ниже показаны основные компоненты EiffelStudio. Рисунок уже приводился в главе 12 при обсуждении сред разработки IDE и EiffelStudio. Чтобы начать использовать EiffelStudio, нет необходимости в чтении этих разделов, но после чтения этой части книги у вас сформируется лучшее понимание, почему вещи устроены именно так, а не иначе.

Е.2. Запуск проекта

В EiffelStudio вы строите системы. Система (называемая также проектом в EiffelStudio) представляет коллекцию классов, сгруппированных для удобства в кластеры. Один из этих классов является корневым; корень определяет точку запуска проекта на выполнение.

В EiffelStudio можно:

- построить полностью новую систему, запустив EiffelStudio, затем выбрав из открывшегося меню File → New project. Мастер будет руководить этим (простым) процессом. Система запускается как обычно – щелчком по соответствующему значку или выбором пункта меню из инсталлированных на компьютере программ. Можно запустить систему и из командной строки;
- продолжить работу с существующей системой. Такая система включает файл «ECF» (Eiffel Control File), автоматически генерируемый в процессе работы сессии EiffelStudio. Этот файл имеет расширение **ecf**; более точно – *s*.**ecf**, где *s* – имя системы. Двойной щелчок по этому файлу (или использование любого другого соглашения, обеспечиваемого вашей операционной системой) будет запускать EiffelStudio на соответствующем проекте.

Если вы хотите работать с одним из примеров этой книги, то можно использовать второй способ, так как соответствующий ECF уже существует. Требуемое соглашение поиска этого файла задано при описании первого примера: каждый пример размещается в подкаталоге каталога примера в поставке Traffic. Имя каждого подкаталога строится из имени главы и имени примера, например, *02_objec*t для системы «*object*», сопровождающей главу 2.

Двойной щелчок ECF в подходящем каталоге запустит EiffelStudio и загрузит систему. После этого первое, что нужно сделать, – это нажать кнопку «Compile», чтобы скомпилировать систему в ее начальной версии. Поработав с начальной версией, можно приступить к ее модификации.

E.3. Показ классов и обликов

Для показа класса в левом верхнем подокне есть несколько возможностей.

- Начать печатать имя класса. Если класс существует, то нет необходимости печатать полное имя, достаточно префикса, а EiffelStudio автоматически дополнит префикс до полного имени. Если класс с указанным именем не существует, EiffelStudio поймет, что вы хотите создать новый класс. Простой мастер помогает в этой работе.
- Используйте технологию «указать и опустить». Щелкните где-либо в интерфейсе имя выбранного класса (или метода, или любого другого программного элемента). Нужно только щелкнуть, кнопку не нужно держать нажатой. После этого передвиньте курсор к подходящей точке, куда можно «опустить» выбранный элемент и снова щелкните правой кнопкой. При правильно выбранном месте, например, в левом верхнем подокне покажется текст выбранного элемента.

Строка кнопок, отображаемая в интерфейсе среды, позволяет изменять формат отображения, выбрав подходящий облик, такой как Contract или Flat (контрактный облик класса или его плоский облик).

Можно отображать как любые облики, так и графическое представление, метрики, информацию компилятора и другие свойства. Для отображения используется нижнее левое подокно и его различные вкладки. Предпочитаемую раскладку окон среды можно сохранить в файле; это полезно, если EiffelStudio используется на разных машинах и вы не хотите повторять процесс установки.

Заметьте: «нижнее левое» и другие позиционные описания ссылаются на раскладку по умолчанию. Внешний вид среды можно настроить по своему вкусу, перетаскивая различные подокна. Раскладка будет сохраняться между сеансами. Если запутаетесь, то всегда можно восстановить раскладку по умолчанию, используя View → Tools Layout → Restore Default Layout.

E.4. Как задать корневой класс и процедуру создания

Для задания или изменения корневого класса и корневой процедуры системы выберите в главном меню пункт File → Project Settings, затем щелкните Target: *s*, где *s* – имя вашей системы. Вход Root укажет корень в формате *root_class.root_procedure*; можно его отредактировать, затем перекомпилировать.

E.5. Управление контрактами

Можно управлять уровнем мониторинга элементов контракта: предусловием, постусловием, инвариантом класса, инвариантами циклов, вариантами циклов и оператором проверки – **check** (последняя конструкция, которая до сих пор не использовалась, – это простой оператор, устанавливающий, что некоторое свойство должно иметь место в данной точке программы). Установки независимо применимы к каждой категории. Чтобы изменить значения, установленные по умолчанию, выберите File → Project Settings, затем **Target**: *s*, где *s* – имя проекта, затем **Assertions**.

Е.6. Управление выполнением и инспекция объектов

Для мониторинга выполнения системы, анализа структуры объектов в период выполнения, пошагового выполнения кода и даже выполнения откатов на несколько шагов назад можно использовать отладчик EiffelStudio. Детальную информацию по его применению можно найти на docs.eiffel.com/book/eiffelstudio/debugger.

Е.7. Состояние паники (ни в коем случае!)

Все должно идти гладко, но если в некоторой точке кажется, что все неожиданно перестало работать, не паникуйте, – помните следующее.

Если проблема в том, что вы не находите инструмент, который ранее в интерфейсе уже использовался, восстановите сохраненную раскладку или раскладку по умолчанию. Используйте для этого, как объяснялось выше, View → Tools Layout.

Если действительно все плохо и вы обнаруживаете, что система в странном состоянии, то причиной может быть повреждение файлов проекта. Тогда вы можете:

- выйти из EiffelStudio;
- перезапустить ее, но напрямую (не щелкая на ECF);
- выбрать File → Open и выбрать ECF вашей системы;
- проверить элемент, помеченный Clean, затем щелкнуть Open. Это откроет систему, но удалит все файлы компиляции. После этого необходимо заново перекомпилировать систему.

Эквивалентный прием, выполняемый вне EiffelStudio, состоит в удалении каталога EIFGEN (для файлов, генерируемых Eiffel) из каталога проекта. Удаленный каталог содержит файлы, сгенерированные компилятором, которые будут созданы при следующей компиляции.

Е.8. Узнать больше

Мощная среда разработки EiffelStudio обладает многими возможностями, лишь первые поверхностные штрихи которых представлены в данном приложении. Полную информацию, включая руководства, учебники, видео, ссылки на документацию, можно найти по адресу: docs.eiffel.com.

Учебное издание

Мейер Бертран

ПОЧУВСТВУЙ КЛАСС

Пер. с англ. под ред. к.т.н. В.А. Биллига

Литературный редактор *С. Перепелкина*
Корректор *О. Ривоненко*
Компьютерная верстка *Н. Овчинникова*

Подписано в печать 25.03.2011. Формат 70x100 $^{1}/_{16}$.
Гарнитура Таймс. Бумага офсетная. Печать офсетная.
Усл. печ. л. 48.5. Тираж 2000 экз. Заказ №

Национальный Открытый Университет «ИНТУИТ»
123056, Москва, Электрический пер., 8, стр. 3.
E-mail: admin@intuit.ru, http://www.intuit.ru

ООО «БИНОМ. Лаборатория знаний»
125167, Москва, проезд Аэропорта, д. 3
Телефон: (499) 157-1902, (499) 157-5272
E-mail: binom@Lbz.ru, http://www.Lbz.ru