Мэтт Вайсфельд

Addison Wesley

Объектно-ориентированное мышление

The Object-Oriented Thought Process

Fourth Edition

Matt Weisfeld

Addison-Wesley

Upper Saddle River, NJ • Boston • Indianapolis • San Francisco
New York • Toronto • Montreal • London • Munich • Paris • Madrid
Cape Town • Sydney • Tokyo • Singapore • Mexico City

ОБЪЕКТНО-ОРИЕНТИРОВАННОЕ МЫШЛЕНИЕ

Мэтт Вайсфельд

ПИТЕР®

Москва · Санкт-Петербург · Нижний Новгород · Воронеж
Ростов-на-Дону · Екатеринбург · Самара · Новосибирск
Киев · Харьков · Минск
2014

ББК 32.988.02-018.1
УДК 004.43
В1

Вайсфельд М.

В1 Объектно-ориентированное мышление. — СПб.: Питер, 2014. — 304 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-496-00793-1

Объектно-ориентированное программирование — это фундамент современных языков программирования, включая C++, Java, C#, Visual Basic, .NET, Ruby и Objective-C. Кроме того, объекты лежат в основе многих веб-технологий, например JavaScript, Python и PHP.

Объектно-ориентированное программирование обеспечивает правильные методики проектирования, переносимость кода и его повторное использование, однако для того, чтобы все это полностью понять, необходимо изменить свое мышление. Разработчики, являющиеся новичками в сфере объектно-ориентированного программирования, не должны поддаваться искушению перейти непосредственно к конкретному языку программирования (например, Objective-C, VB .NET, C++, C#, .NET или Java) или моделирования (например, UML), а вместо этого сначала уделить время освоению того, что автор книги Мэтт Вайсфельд называет объектно-ориентированным мышлением.

Несмотря на то что технологии программирования изменяются и эволюционируют с годами, объектно-ориентированные концепции остаются прежними — при этом не важно, какой именно является платформа.

12+ (В соответствии с Федеральным законом от 29 декабря 2010 г. № 436-ФЗ.)

ББК 32.988.02-018.1
УДК 004.43

ISBN 978-0321861276 англ.
ISBN 978-5-496-00793-1

Краткое содержание

Оглавление

Эта книга посвящается Шэрон,
Стейси, Стефани и Даффи

Об авторе

Мэтт Вайсфельд (Matt Weisfeld) — профессор колледжа, разработчик программного обеспечения и писатель. Он проживает в Кливленде, штат Огайо. Предже чем стать штатным преподавателем в колледже, Мэтт 20 лет проработал в индустрии информационных технологий. У него есть степень магистра наук в информатике и магистра делового администрирования. Помимо первых трех изданий книги «Объектно-ориентированное мышление», он написал две другие на тему разработки программного обеспечения, а также опубликовал множество статей в таких журналах, как *developer.com*, *Dr. Dobb's Journal*, *The C/C++ Users Journal*, *Software Development Magazine*, *Java Report* и международный журнал *Project Management*.

Благодарности

Как и в случае с тремя первыми изданиями, создание этой книги потребовало совместных усилий многих людей. Я хотел бы уделить время тому, чтобы поблагодарить как можно больше этих людей, поскольку без их помощи данная книга никогда бы не увидела свет.

Прежде всего мне хотелось бы выразить благодарность моей жене Шэрон за всю ее помощь. Она не только поддерживала и подбадривала меня во время длительного написания книги, но и выступила в роли первого редактора всех моих письменных материалов.

Я также хотел бы поблагодарить мою маму и остальных членов семьи за их постоянную поддержку.

Трудно поверить в то, что работа над первым изданием этой книги началась еще в 1998 году. На протяжении всех этих лет я всесторонне наслаждался сотрудничеством со всеми людьми из компании Pearson, трудясь над четырьмя изданиями. Работа с такими редакторами, как Марк Тэбер (Mark Taber), Сонглин Киу (Songlin Qiu), Барбара Хача (Barbara Hacha) и Сет Керни (Seth Kerney), была для меня удовольствием.

Выражаю особую благодарность Джону Апчерчу (Jon Upchurch) за его проверку значительной части приведенного в этой книге кода, а также за техническое редактирование рукописи. Его владение удивительно широким диапазоном технических тем очень помогло мне.

Мне хотелось бы поблагодарить Донни Сантоса (Donnie Santos) за его знания в области разработки мобильных и гибридных приложений, а также Objective-C.

И наконец, спасибо моим дочерям Стейси и Стефани и моему коту Даффи за то, что они постоянно держат меня в форме.

От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты vinitski@minsk.piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На сайте издательства http://www.piter.com вы найдете подробную информацию о наших книгах.

Введение

Как следует из названия, эта книга посвящена объектно-ориентированному мышлению. Хотя выбор темы и названия книги является важным решением, оно оказывается совсем не простым, когда речь идет о концептуальной тематике. Во многих книгах рассматривается тот или иной уровень программирования и объектной ориентации. В отдельных популярных книгах охватываются темы, в число которых входят объектно-ориентированный анализ, объектно-ориентированное проектирование, шаблоны проектирования, объектно-ориентированные данные (XML), унифицированный язык моделирования Unified Modeling Language (UML), объектно-ориентированная веб-разработка (в том числе мобильная), различные объектно-ориентированные языки программирования и многие другие темы, связанные с объектно-ориентированным программированием (ООП).

Однако при чтении таких книг многие люди забывают, что все эти темы базируются на одном фундаменте: важно, как вы мыслите объектно-ориентированным образом. Зачастую бывает так, что многие профессионалы в области создания программного обеспечения, а также учащиеся начинают читать эти книги, не потратив достаточно времени и усилий на то, чтобы *действительно* разобраться в концепциях проектирования, кроющихся за кодом.

Я считаю, что освоение объектно-ориентированных концепций не сводится к изучению конкретного метода разработки, языка программирования или набора инструментов проектирования. Работа в объектно-ориентированном стиле, попросту говоря, является образом мышления. Эта книга всецело посвящена объектно-ориентированному мышлению.

Отделение языков программирования, методик и инструментов разработки от объектно-ориентированного мышления — нелегкая задача. Зачастую люди знакомятся с объектно-ориентированными концепциями, «ныряя» с головой в тот или иной язык программирования. Например, какое-то время назад многие программисты на C впервые столкнулись с объектной ориентацией, перейдя прямо на C++ еще до того, как они хотя бы отдаленно познакомились с объектно-ориентированными концепциями. Первое знакомство других профессионалов в области создания программного обеспечения с объектной ориентацией произошло в контексте представлений, которые включали объектные модели с использованием UML, — опять-таки до того, как эти разработчики хотя бы соприкоснулись непосредственно с объектно-ориентированными концепциями. Даже сейчас, спустя пару десятилетий после того как Интернет стал использоваться в качестве бизнес-платформы, нередко можно увидеть книги и профессиональные учебные материалы по программированию, в которых обсуждение объектно-ориентированных концепций откладывается на потом.

Важно понимать значительную разницу между изучением объектно-ориентированных концепций и программированием на объектно-ориентированном языке. Я четко осознал это до того как начал работать еще над первым изданием данной

книги, когда прочитал статьи вроде написанной Крейгом Ларманом (Craig Larman) под названием *What the UML Is — and Isn't* («Что такое UML и чем он не является»). В этой статье он пишет:

> «К сожалению, в контексте разработки программного обеспечения и языка UML, позволяющего создавать диаграммы, умение читать и писать UML-нотацию, похоже, иногда приравнивается к навыкам объектно-ориентированного анализа и проектирования. Конечно, на самом деле это не так, и последнее из упомянутого намного важнее, чем первое. Поэтому я рекомендую искать учебные курсы и материалы, в которых приобретению интеллектуальных навыков в объектно-ориентированном анализе и проектировании придается первостепенное значение по сравнению с написанием UML-нотации или использованием средств автоматизированной разработки программного обеспечения».

Таким образом, несмотря на то что изучение языка моделирования является важным шагом, намного важнее сначала приобрести объектно-ориентированные навыки. Изучение UML до того как вы полностью разберетесь в объектно-ориентированных концепциях, аналогично попытке научиться читать электрические схемы, изначально ничего не зная об электричестве.

Такая же проблема наблюдается с языками программирования. Как уже отмечалось, многие программисты на C ступили на территорию объектной ориентации, перейдя на C++ прежде, чем соприкоснуться с объектно-ориентированными концепциями. Это всегда обнаруживается в процессе собеседования. Во многих случаях разработчики, утверждающие, что они являются программистами на C++, в действительности оказываются просто программистами на C, использующими компиляторы C++. Даже сейчас, когда такие языки, как C# .NET, VB .NET, Objective-C и Java, стали общепринятыми, несколько ключевых вопросов во время собеседования при приеме на работу могут быстро выявить сложности в понимании объектно-ориентированных концепций.

Ранние версии Visual Basic не являются объектно-ориентированными. Язык C тоже не объектно-ориентированный, а C++ *разрабатывался* как обратно совместимый с ним. По этой причине вполне возможно использовать компилятор C++ при написании синтаксиса на C, но отказавшись от всех объектно-ориентированных свойств C++. Objective-C создавался как расширение стандартного языка ANSI C. Что еще хуже, программисты могут применять ровно столько объектно-ориентированных функций, сколько нужно для того, чтобы сделать приложения непонятными для остальных программистов, как использующих, так и не использующих объектно-ориентированные языки.

Таким образом, жизненно важно, чтобы вы, приступая к изучению того как пользоваться средами объектно-ориентированной разработки, сначала освоили фундаментальные объектно-ориентированные концепции. Не поддавайтесь искушению сразу перейти непосредственно к языку программирования (например, Objective-C, VB .NET, C++, C# .NET или Java) или языку моделирования (например, UML), а вместо этого сначала уделите время освоению объектно-ориентированного мышления.

На протяжении многих лет я программировал на C, а в конце 1980-х годов начал изучать Smalltalk. Компания, в которой я работал на тот момент, решила, что ее

разработчикам программного обеспечения необходимо изучить эту перспективную технологию. Преподаватель начал занятия, сказав, что объектно-ориентированная парадигма представляет собой совершенно новый образ мышления *(несмотря на тот факт, что она существует еще с 1960-х годов)*. Затем он сказал, что, хотя все мы, скорее всего, очень хорошие программисты, 10–20 % из нас никогда по-настоящему не поймут объектно-ориентированный подход. Если это заявление и является действительно верным, то, наиболее вероятно, по той причине, что некоторые хорошие программисты никогда не тратят время на смену парадигмы, а изучают основополагающие объектно-ориентированные концепции.

Что нового в четвертом издании

Как часто отмечалось во введении к этой книге, мое видение первого издания заключалось в том, чтобы придерживаться концепций, а не сосредотачиваться на конкретной новейшей технологии. Несмотря на следование такому же подходу во втором, третьем и четвертом изданиях, я включил главы, посвященные нескольким программным темам, которые хорошо согласуются с объектно-ориентированными концепциями. В главах 1–10 рассматриваются фундаментальные объектно-ориентированные концепции, а в главах 11–15 внимание сосредоточено на применении этих концепций к некоторым общим объектно-ориентированным технологиям. Так, например, главы 1–10 обеспечивают фундамент для курса по основным принципам объектно-ориентированного программирования (таким так инкапсуляция, полиморфизм, наследование и т. п.), а в главах 11–15 рассматриваются вопросы их практического применения.

В четвертом издании я подробнее изложил многие темы из предыдущих изданий. Пересмотрены и обновлены материалы на такие темы, как:

- разработка программ для мобильных устройств, в том числе приложений для телефонов, мобильных приложений, мобильных веб-приложений, гибридных приложений и т. д.;
- примеры кода на Objective-C с целью охвата среды iOS;
- удобный для восприятия обмен данными с использованием XML и JSON;
- отображение и преобразование данных с применением CSS, XSLT и т. д.;
- веб-службы, включая простой протокол доступа к объектам Simple Object Access Protocol (SOAP), веб-службы RESTful и т. п.;
- клиентские/серверные технологии и маршалинг объектов;
- постоянные данные и сериализация объектов;
- расширенные примеры кода для определенных глав на Java, C# .NET, VB .NET и Objective-C, доступные в режиме онлайн на сайте издателя.

Целевая аудитория

Эта книга представляет собой общее введение в фундаментальные объектно-ориентированные концепции и содержит примеры кода, которые их подкрепляют.

Одним из самых сложных трюков оказалось сохранение концептуальности кода с одновременным обеспечением твердой кодовой базы. Цель этой книги в том, чтобы читатели смогли понять соответствующие концепции и технологию без необходимости использовать компилятор. Однако если он окажется в вашем распоряжении, то здесь вы найдете код для выполнения и исследования.

Целевая аудитория этой книги включает проектировщиков, разработчиков, программистов, менеджеров проектов и всех, кто хочет получить общее представление о том, что такое объектная ориентация. Прочтение этой книги должно обеспечить прочный фундамент для перехода к изучению материала других изданий, в которых рассматриваются более современные объектно-ориентированные темы.

Среди таких более продвинутых книг одной из моих любимых является «Объектно-ориентированное проектирование на Java» *(Object-Oriented Design in Java)* под авторством Стивена Гилберта (Stephen Gilbert) и Билла Маккарти (Bill McCarty). Мне очень нравится подход, использованный в этом издании, выступившем для меня в качестве руководства, когда я преподавал объектно-ориентированные концепции. По ходу всей своей книги я часто ссылаюсь на «Объектно-ориентированное проектирование на Java» и рекомендую вам прочитать его после того, как закончите читать эту книгу.

К числу прочих книг, которые я считаю очень полезными, относятся «Эффективное использование C++» *(Effective C++)* Скотта Майерса (Scott Meyers), «Классическая и объектно-ориентированная разработка программного обеспечения» *(Classical and Object-Oriented Software Engineering)* Стивена Р. Шача (Stephen R. Schach), «Философия C++» *(Thinking in C++)* Брюса Эккеля (Bruce Eckel), «UML. Основы» *(UML Distilled)* Мартина Фаулера (Martin Fowler) и «Проектирование на Java» *(Java Design)* Петера Коуда (Peter Coad) и Марка Мейфилда (Mark Mayfield).

Во время обучения программистов основам программирования и веб-разработке в корпорациях и университетах я быстро осознал, что большинство этих людей легко схватывали синтаксис языка, но в то же время им непросто давалась объектно-ориентированная природа языка.

Подход, использованный в этой книге

К настоящему времени должна быть очевидна моя твердая убежденность в том, что сначала нужно хорошо освоить объектно-ориентированное мышление, а затем уже приступать к изучению языка программирования или моделирования. Эта книга наполнена примерами кода и UML-диаграмм, однако вам не обязательно владеть определенным языком программирования или UML для того, чтобы переходить к ее чтению. Но после всего того, что я сказал об изучении в первую очередь объектно-ориентированных концепций, почему же в этой книге так много кода на Java, C# .NET, VB .NET и Objective-C и столь большое количество UML-диаграмм? Во-первых, они отлично иллюстрируют объектно-ориентированные концепции. Во-вторых, они жизненно важны для освоения объектно-ориентированного мышления и должны рассматриваться на вводном уровне. Основной принцип заключается не в том, чтобы сосредотачиваться на Java, C# .NET, VB .NET, Objective-C или

UML, а в использовании их в качестве средств, которые помогают понять основополагающие концепции.

Обратите внимание на то, что мне очень нравится применять UML-диаграммы классов как визуальные средства, помогающие понять классы, их атрибуты и методы. Фактически диаграммы классов — это единственный компонент UML, использованный в этой книге. Я считаю, что UML-диаграммы классов отлично подходят для представления концептуальной природы объектных моделей. Я продолжу использовать объектные модели в качестве образовательного инструмента для наглядной демонстрации конструкции классов и того, как классы соотносятся друг с другом.

Примеры кода в этой книге иллюстрируют концепции вроде циклов и функций. Однако понимание этого кода как такового не является необходимым условием для понимания самих концепций; возможно, целесообразно иметь под рукой книгу, в которой рассматривается синтаксис соответствующего языка, если вы захотите узнать дополнительные подробности.

Я не могу слишком строго утверждать, что эта книга *не* учит языку Java, C# .NET, VB .NET, Objective-C или UML, каждому из которых можно было бы посвятить целые тома. Я надеюсь, что она пробудит в вас интерес к другим объектно-ориентированным темам вроде объектно-ориентированного анализа, объектно-ориентированного проектирования и объектно-ориентированного программирования.

Соглашения, принятые в книге

В этой книге приняты следующие соглашения:

- строки кода, команды, операторы и другие связанные с кодом элементы оформлены с применением `моноширинного` шрифта;
- по ходу всей книги имеются специальные врезки на страницах вроде тех, что показаны далее.

СОВЕТ

В такой врезке приводится рекомендация или демонстрируется легкий способ сделать что-то.

ПРИМЕЧАНИЕ

В этой врезке приводится интересная информация, связанная с рассматриваемым вопросом, — немного более подробные сведения или указатель на какую-то новую методику.

ПРЕДОСТЕРЕЖЕНИЕ

В такой врезке содержится предупреждение о некой возможной проблеме и дается совет, как ее избежать.

Глава 1
Введение в объектно-ориентированные концепции

Хотя многие программисты не осознают этого, объектно-ориентированная разработка программного обеспечения существует с начала 1960-х годов. Только во второй половине 1990-х годов объектно-ориентированная парадигма начала набирать обороты, несмотря на тот факт, что популярные объектно-ориентированные языки программирования вроде Smalltalk и C++ уже широко использовались.

Расцвет объектно-ориентированных технологий совпал с началом применения Интернета в качестве платформы для бизнеса и развлечений. А после того как стало очевидным, что Глобальная сеть активно проникает в жизнь людей, объектно-ориентированные технологии уже заняли удобную позицию для того, чтобы помочь в разработке новых веб-технологий.

Важно подчеркнуть, что название этой главы звучит как «Введение в объектно-ориентированные концепции». В качестве ключевого здесь использовано слово «концепции», а не «технологии». Технологии в индустрии программного обеспечения очень быстро изменяются, в то время как концепции эволюционируют. Я использовал термин «эволюционируют» потому, что, хотя концепции остаются относительно устойчивыми, они все же претерпевают изменения. Это очень интересная особенность, заметная при тщательном изучении концепций. Несмотря на их устойчивость, они постоянно подвергаются повторным интерпретациям, а это предполагает весьма любопытные дискуссии.

Эту эволюцию можно легко проследить за последние два десятка лет, если понаблюдать за прогрессом различных индустриальных технологий, начиная с первых примитивных браузеров второй половины 1990-х годов и заканчивая мобильными/телефонными/веб-приложениями, доминирующими сегодня. Как и всегда, новые разработки окажутся не за горами, когда мы будем исследовать гибридные приложения и пр. На всем протяжении путешествия объектно-ориентированные концепции присутствовали на каждом этапе. Вот почему вопросы, рассматриваемые в этой главе, так важны. Эти концепции сегодня так же актуальны, как и 20 лет назад.

Фундаментальные концепции

Основная задача этой книги — заставить вас задуматься о том, как концепции используются при проектировании объектно-ориентированных систем. Исторически

сложилось так, что объектно-ориентированные языки определяются следующими концепциями: *инкапсуляцией*, *наследованием* и *полиморфизмом*. Поэтому если тот или иной язык программирования не реализует все эти концепции, то он, как правило, не считается объектно-ориентированным. Наряду с этими тремя терминами я всегда включаю в общую массу композицию; таким образом, мой список объектно-ориентированных концепций выглядит так:

- инкапсуляция;
- наследование;
- полиморфизм;
- композиция.

Мы детально рассмотрим все эти концепции в данной книге.

Одна из трудностей, с которыми мне пришлось бороться еще с самого первого издания книги, заключается в том, как эти концепции соотносятся непосредственно с текущими методиками проектирования, ведь они постоянно изменяются. Например, все время ведутся дебаты об использовании наследования при объектно-ориентированном проектировании. Нарушает ли наследование инкапсуляцию на самом деле? (Эта тема будет рассмотрена в последующих главах.) Даже сейчас многие разработчики стараются избегать наследования, насколько это представляется возможным.

Мой подход, как и всегда, состоит в том, чтобы придерживаться концепций. Независимо от того, будете вы использовать наследование или нет, вам как минимум потребуется понять, что такое наследование, благодаря чему вы сможете сделать обоснованный выбор методики проектирования. Как уже отмечалось во введении к этой книге, ее целевой аудиторией являются люди, которым требуется *общее введение в фундаментальные объектно-ориентированные концепции*. Исходя из этой формулировки в текущей главе я представляю фундаментальные объектно-ориентированные концепции с надеждой обеспечить моим читателям твердую основу для принятия важных решений касательно проектирования. Рассматриваемые здесь концепции затрагивают большинство, если не все темы, охватываемые в последующих главах, в которых соответствующие вопросы исследуются намного подробнее.

Объекты и унаследованные системы

По мере того как объектно-ориентированное программирование получало широкое распространение, одна из проблем, с которыми сталкивались разработчики, заключалась в интеграции объектно-ориентированных технологий с существующими системами. В то время разграничивались объектно-ориентированное и структурное (или процедурное) программирование, которое было доминирующей парадигмой разработки на тот момент. Мне всегда это казалось странным, поскольку, на мой взгляд, объектно-ориентированное и структурное программирование не конкурируют друг с другом. Они являются взаимодополняющими, так как объекты хорошо интегрируются со структурированным кодом. Даже сейчас я часто слышу такой вопрос: «Вы занимаетесь структурным или объектно-ориентированным программированием?» Недолго думая, я бы ответил: «И тем, и другим».

В том же духе объектно-ориентированный код не призван заменить структурированный код. Многие не являющиеся объектно-ориентированными *унаследованные системы* (то есть более старые по сравнению с уже используемыми) довольно хорошо справляются со своей задачей. Зачем же тогда идти на риск столкнуться с возможными проблемами, изменяя или заменяя эти унаследованные системы? В большинстве случаев вам не следует изменять их, по крайней мере не ради лишь внесения изменений. В сущности, в системах, основанных не на объектно-ориентированном коде, нет ничего плохого. Однако совершенно новые разработки, несомненно, подталкивают задуматься об использовании объектно-ориентированных технологий (в некоторых случаях нет иного выхода, кроме как поступить именно так).

Хотя на протяжении последних 20 лет наблюдалось постоянное и значительное увеличение количества объектно-ориентированных разработок, зависимость мирового сообщества от сетей вроде Интернета и мобильных инфраструктур поспособствовала еще более широкому их распространению. Буквально взрывной рост количества транзакций, осуществляемых в браузерах и мобильных приложениях, открыл совершенно новые рынки, где значительная часть разработок программного обеспечения была новой и главным образом не обремененной заботами, связанными с унаследованными системами. Но даже если вы все же столкнетесь с такими заботами, то на этот случай есть тенденция, согласно которой унаследованные системы можно заключать в *объектные обертки*.

ОБЪЕКТНЫЕ ОБЕРТКИ

Объектные обертки представляют собой объектно-ориентированный код, в который заключается другой код. Например, вы можете взять структурированный код (вроде циклов и условий) и заключить его в объект, чтобы этот код выглядел как объект. Вы также можете использовать объектные обертки для заключения в них функциональности, например параметров, касающихся безопасности, или непереносимого кода, связанного с аппаратным обеспечением, и т. д. Обертывание структурированного кода детально рассматривается в главе 6.

Сегодня одной из наиболее интересных областей разработки программного обеспечения является интеграция унаследованного кода с мобильными и веб-системами. Во многих случаях мобильное клиентское веб-приложение в конечном счете «подключается» к данным, располагающимся на мейнфрейме. Разработчики, одновременно обладающие навыками в веб-разработке как для мейнфреймов, так и для мобильных устройств, весьма востребованны.

Вы сталкиваетесь с объектами в своей повседневной жизни, вероятно, даже не осознавая этого. Вы можете столкнуться с ними, когда едете в своем автомобиле, разговариваете по сотовому телефону, используете свою домашнюю развлекательную систему, играете в компьютерные игры, а также во многих других ситуациях. Электронные соединения по сути превратились в соединения, основанные на объектах. Тяготея к мобильным веб-приложениям, бизнес тяготеет к объектам, поскольку технологии, используемые для электронной торговли, по своей природе в основном являются объектно-ориентированными.

МОБИЛЬНАЯ ВЕБ-РАЗРАБОТКА

Несомненно, появление Интернета значительно поспособствовало переходу на объектно-ориентированные технологии. Дело в том, что объекты хорошо подходят для использования в сетях. Хотя Интернет был в авангарде этой смены парадигмы, мобильные сети теперь заняли не последнее место в общей массе. В этой книге термин «мобильная веб-разработка» будет использоваться в контексте концепций, которые относятся как к разработке мобильных веб-приложений, так и к веб-разработке. Термин «гибридные приложения» иногда будет применяться для обозначения приложений, которые работают в браузерах как на веб-устройствах, так и на мобильных аппаратах.

Процедурное программирование в сравнении с объектно-ориентированным

Прежде чем мы углубимся в преимущества объектно-ориентированной разработки, рассмотрим более существенный вопрос: что именно такое объект? Это одновременно и сложный, и простой вопрос. Сложный он потому, что изучение любого метода разработки программного обеспечения не является тривиальным. А простой он в силу того, что люди уже мыслят объектно.

Например, когда вы смотрите на какого-то человека, вы видите его как объект. При этом объект определяется двумя компонентами: атрибутами и поведением. У человека имеются такие атрибуты, как цвет глаз, возраст, вес и т. д. Человек также обладает поведением, то есть он ходит, говорит, дышит и т. д. В соответствии со своим базовым определением *объект* — это сущность, *одновременно* содержащая данные и поведения.

Слово *«одновременно»* в данном случае определяет ключевую разницу между объектно-ориентированным программированием и другими методологиями программирования. Например, при процедурном программировании код размещается в полностью отдельных функциях или процедурах. В идеале, как показано на рис. 1.1, эти процедуры затем превращаются в «черные ящики», куда поступают входные данные и откуда потом выводятся выходные данные. Данные размещаются в отдельных структурах, а манипуляции с ними осуществляются с помощью этих функций или процедур.

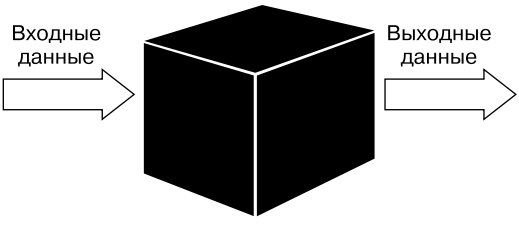

Рис. 1.1. Черный ящик

РАЗНИЦА МЕЖДУ ОБЪЕКТНО-ОРИЕНТИРОВАННЫМ И СТРУКТУРНЫМ ПРОЕКТИРОВАНИЕМ

При объектно-ориентированном проектировании атрибуты и поведения размещаются в рамках одного объекта, в то время как при процедурном или структурном проектировании атрибуты и поведения обычно разделяются.

При росте популярности объектно-ориентированного проектирования один из фактов, который изначально тормозил его принятие людьми, заключался в том, что использовалось много систем, которые не являлись объектно-ориентированными, но отлично работали. Таким образом, с точки зрения бизнеса не было никакого смысла изменять эти системы лишь ради внесения изменений. Каждому, кто знаком с любой компьютерной системой, известно, что то или иное изменение может привести к катастрофе, даже если предполагается, что это изменение будет незначительным.

В то же время люди не принимали объектно-ориентированные базы данных. В определенный момент при появлении объектно-ориентированной разработки в какой-то степени вероятным казалось то, что такие базы данных смогут заменить реляционные базы данных. Однако этого так никогда и не произошло. Бизнес вложил много денег в реляционные базы данных, а совершению перехода препятствовал главный фактор — они работали. Когда все издержки и риски преобразования систем из реляционных баз данных в объектно-ориентированные стали очевидными, неоспоримых доводов в пользу перехода не оказалось.

На самом деле бизнес сейчас нашел золотую середину. Для многих методик разработки программного обеспечения характерны свойства объектно-ориентированной и структурной методологий разработки.

Как показано на рис. 1.2, при структурном программировании данные зачастую отделяются от процедур и являются глобальными, благодаря чему их легко модифицировать вне области видимости вашего кода. Это означает, что доступ к данным неконтролируемый и непредсказуемый (то есть у множества функций может быть доступ к глобальным данным). Во-вторых, поскольку у вас нет контроля над тем, кто сможет получить доступ к данным, тестирование и отладка намного усложняются. При работе с объектами эта проблема решается путем объединения данных и поведения в рамках одного элегантного полного пакета.

ПРАВИЛЬНОЕ ПРОЕКТИРОВАНИЕ

Мы можем сказать, что при правильном проектировании в объектно-ориентированных моделях нет такого понятия, как глобальные данные. По этой причине в объектно-ориентированных системах обеспечивается высокая степень целостности данных.

Вместо того чтобы заменять другие парадигмы разработки программного обеспечения, объекты стали эволюционной реакцией. Структурированные программы содержат комплексные структуры данных вроде массивов и т. д. C++ включает структуры, которые обладают многими характеристиками объектов (классов).

Однако объекты представляют собой нечто намного большее, чем структуры данных и примитивные типы вроде целочисленных и строковых. Хотя объекты содержат такие сущности, как целые числа и строки, используемые для представления атрибутов, они также содержат методы, которые характеризуют поведение. В объектах методы применяются для выполнения операций с данными, а также

для совершения других действий. Пожалуй, более важно то, что вы можете управлять доступом к членам объектов (как к атрибутам, так и к методам). Это означает, что отдельные из этих членов можно скрыть от других объектов. Например, объект с именем `Math` может содержать две целочисленные переменные с именами `myInt1` и `myInt2`. Скорее всего, объект `Math` также содержит методы, необходимые для извлечения значений `myInt1` и `myInt2`. Он также может включать метод с именем `sum()` для сложения двух целочисленных значений.

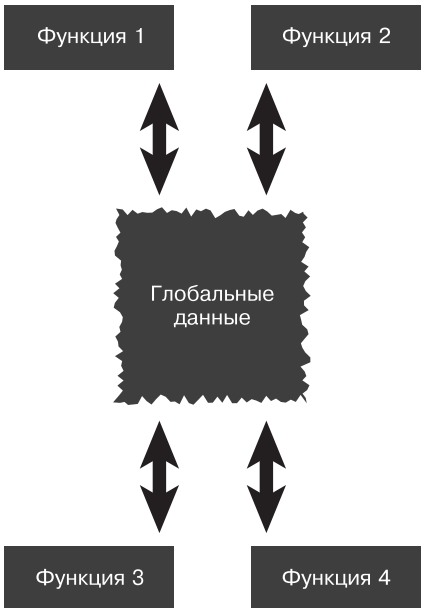

Рис. 1.2. Использование глобальных данных

СКРЫТИЕ ДАННЫХ

В объектно-ориентированной терминологии данные называются атрибутами, а поведения — методами. Ограничение доступа к определенным атрибутам и/или методам называется скрытием данных.

Объединив атрибуты и методы в одной сущности (это действие называется *инкапсуляцией*), мы можем управлять доступом к данным в объекте `Math`. Если определить целочисленные переменные `myInt1` и `myInt2` в качестве «запретной зоны», то другая логически несвязанная функция не будет иметь возможности осуществлять манипуляции с ними, и только объект `Math` сможет делать это.

РУКОВОДСТВО ПО ПРОЕКТИРОВАНИЮ КЛАССОВ SOUND

Можно создать неудачно спроектированные объектно-ориентированные классы, которые не ограничивают доступ к атрибутам классов. Суть в том, что при объектно-ориентированном проектировании вы можете создать плохой код с той же легкостью, как и при использовании любой другой методологии программирования. Просто примите меры для того, чтобы придерживаться руководства по проектированию классов Sound (см. главу 5).

А что будет, если другому объекту — например, `myObject` — потребуется получить доступ к сумме значений `myInt1` и `myInt2`? Он обратится к объекту `Math`: `myObject` отправит сообщение объекту `Math`. На рис. 1.3 показано, как два объекта общаются друг с другом с помощью своих методов. Сообщение на самом деле представляет собой вызов метода `sum` объекта `Math`. Метод `sum` затем возвращает значение объекту `myObject`. Вся прелесть заключается в том, что `myObject` не нужно знать, как вычисляется сумма (хотя, я уверен, он может догадаться). Используя эту методологию проектирования, вы можете изменить то, как объект `Math` вычисляет сумму, не меняя объекта `myObject` (при условии, что средства для извлечения значения суммы останутся прежними). Все, что вам нужно, — это сумма, и вам *безразлично*, как она вычисляется.

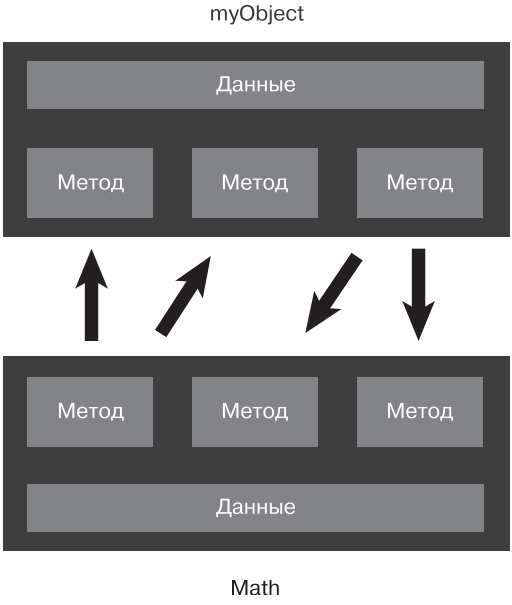

Рис. 1.3. Коммуникации между объектами

Простой пример с калькулятором позволяет проиллюстрировать эту концепцию. При определении суммы на калькуляторе вы используете только его интерфейс — кнопочную панель и экран на светодиодах. В калькулятор заложен метод для вычисления суммы, который вызывается, когда вы нажимаете соответствующую последовательность кнопок. После этого вы сможете получить правильный ответ, однако не будете знать, как именно этот результат был достигнут, — ни электронно, ни алгоритмически.

Вычисление суммы не является обязанностью объекта `myObject` — она возлагается на `Math`. Пока у `myObject` есть доступ к объекту `Math`, он сможет отправлять соответствующие сообщения и получать надлежащие результаты. Вообще говоря, объекты не должны манипулировать внутренними данными других объектов (то есть `myObject` не должен напрямую изменять значения `myInt1` и `myInt2`). Кроме

того, по некоторым причинам (их мы рассмотрим позднее) обычно лучше создавать небольшие объекты со специфическими задачами, нежели крупные, но выполняющие много задач.

Переход с процедурной разработки на объектно-ориентированную

Теперь, когда мы имеем общее понятие о некоторых различиях между процедурными и объектно-ориентированными технологиями, сильнее углубимся и в те и в другие.

Процедурное программирование

При процедурном программировании данные той или иной системы обычно отделяются от операций, используемых для манипулирования ими. Например, если вы решите передать информацию по сети, то будут отправлены только релевантные данные (рис. 1.4) с расчетом на то, что программа на другом конце сетевой магистрали будет знать, что с ними делать. Иными словами, между клиентом и сервером должно быть заключено что-то вроде джентльменского соглашения для передачи данных. При такой модели вполне возможно, что на самом деле по сети не будет передаваться никакого кода.

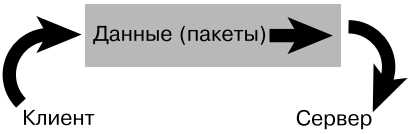

Рис. 1.4. Данные, передаваемые по сети

Объектно-ориентированное программирование

Основное преимущество объектно-ориентированного программирования заключается в том, что и данные, и операции (код), используемые для манипулирования ими, инкапсулируются в одном объекте. Например, при перемещении объекта по сети он передается целиком, включая данные и поведение.

ЕДИНОЕ ЦЕЛОЕ

Хотя мышление в контексте единого целого теоретически является прекрасным подходом, сами поведения не получится отправить из-за того, что с обеих сторон имеются копии соответствующего кода. Однако важно мыслить в контексте всего объекта, передаваемого по сети в виде единого целого.

На рис. 1.5 показана передача объекта `Employee` по сети.

ПРАВИЛЬНОЕ ПРОЕКТИРОВАНИЕ

Хорошим примером этой концепции является объект, загружаемый браузером. Часто бывает так, что браузер заранее не знает, какие действия будет выполнять определенный объект, поскольку он еще «не видел» кода. Когда объект загрузится, браузер выполнит код, содержащийся в этом объекте, а также использует заключенные в нем данные.

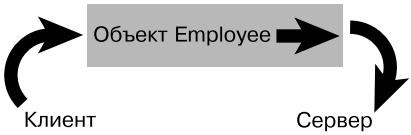

Рис. 1.5. Объект, передаваемый по сети

Что такое объект

Объекты — это строительные блоки объектно-ориентированных программ. Та или иная программа, которая задействует объектно-ориентированную технологию, по сути является набором объектов. В качестве наглядного примера рассмотрим корпоративную систему, содержащую объекты, которые представляют собой работников соответствующей компании. Каждый из этих объектов состоит из данных и поведений, описанных в последующих разделах.

Данные объектов

Данные, содержащиеся в объекте, представляют его состояние. В терминологии объектно-ориентированного программирования эти данные называются *атрибутами*. В нашем примере, как показано на рис. 1.6, атрибутами работника могут быть номер социального страхования, дата рождения, пол, номер телефона и т. д. Атрибуты включают информацию, которая разнится от одного объекта к другому (ими в данном случае являются работники). Более подробно атрибуты рассматриваются далее в этой главе при исследовании классов.

Рис. 1.6. Атрибуты объекта Employee

Поведения объектов

Поведение объекта представляет то, что он может сделать. В процедурных языках поведение определяется процедурами, функциями и подпрограммами. В терминологии объектно-ориентированного программирования поведения объектов содер-

жатся в *методах*, а вызов метода осуществляется путем отправки ему сообщения. Примите по внимание, что в нашем примере с работниками одно из необходимых поведений объекта `Employee` заключается в задании и возврате значений различных атрибутов. Таким образом, у каждого атрибута будут иметься соответствующие методы, например `setGender()` и `getGender()`. В данном случае, когда другому объекту потребуется такая информация, он сможет отправить сообщение объекту `Employee` и узнать значение его атрибута `gender`.

Неудивительно, что применение геттеров и сеттеров, как и многое из того, что включает объектно-ориентированная технология, эволюционировало с тех пор, как было опубликовано первое издание этой книги. Это особенно актуально для тех случаев, когда дело касается данных. Как мы еще увидим в главах 11 и 12, сейчас данные конструируются объектно-ориентированным образом. Помните, что одно из самых интересных преимуществ использования объектов заключается в том, что данные являются частью пакета — они не отделяются от кода.

Появление XML не только сосредоточило внимание людей на представлении данных в переносимом виде, но и обеспечило для кода альтернативные способы доступа к данным. В .NET-методиках геттеры и сеттеры считаются свойствами самих данных.

Например, взгляните на атрибут с именем `Name`, который при использовании в Java выглядит следующим образом:

```
public String Name;
```

Соответствующие геттер и сеттер выглядели бы так:

```
public void setName (String n) {name = n;};
public String getName() {return name;};
```

Теперь, при создании XML-атрибута с именем `Name`, определение на C# .NET может выглядеть примерно так:

```
Private string strName;
public String Name
{
  get
  {
    return this.strName;
  }
  set
  {
    if (value == null) return;
    this.strName = value;
  }
}
```

При таком подходе геттеры и сеттеры в действительности являются *свойствами* атрибутов — в данном случае атрибута с именем `Name`.

Независимо от используемого подхода цель одна и та же — управляемый доступ к атрибуту. В этой главе я хочу сначала сосредоточиться на концептуальной природе

методов доступа. О свойствах мы поговорим подробнее, когда будем рассматривать объектно-ориентированные данные в главе 11 и последующих.

ГЕТТЕРЫ И СЕТТЕРЫ

Концепция геттеров и сеттеров поддерживает концепцию скрытия данных. Поскольку другие объекты не должны напрямую манипулировать данными, содержащимися в одном из объектов, геттеры и сеттеры обеспечивают управляемый доступ к данным объекта. Геттеры и сеттеры иногда называют методами доступа и методами-модификаторами соответственно.

Следует отметить, что мы показываем только интерфейс методов, а не реализацию. Приведенная далее информация — это все, что пользователям потребуется знать для эффективного применения методов:

- имя метода;
- параметры, передаваемые методу;
- возвращаемый тип метода.

Поведения показаны на рис. 1.7.

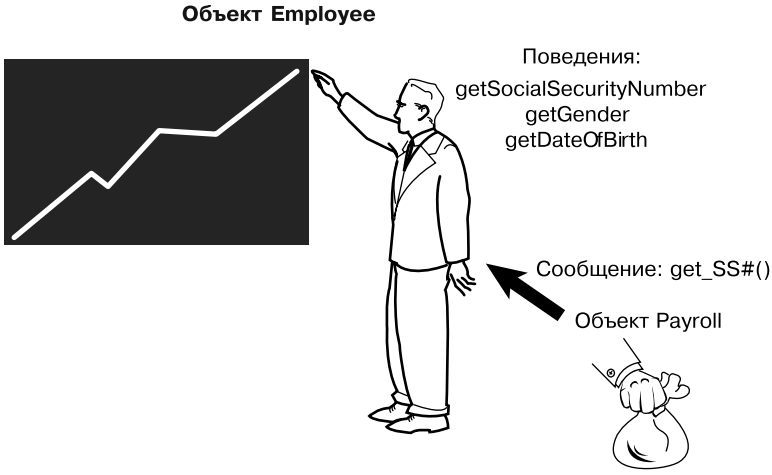

Рис. 1.7. Поведения объекта Employee

На рис. 1.7 демонстрируется, что объект `Payroll` содержит метод с именем `calculatePay()`, который используется для вычисления суммы зарплаты каждого конкретного работника. Помимо прочей информации, объекту `Payroll` потребуется номер социального страхования соответствующего работника. Для этого он должен отправить сообщение объекту `Employee` (в данном случае дело касается метода `getSocialSecurityNumber()`). В сущности, это означает, что объект `Payroll` вызовет метод `getSocialSecurityNumber()` объекта `Employee`. Объект `Employee` «увидит» это сообщение и возвратит запрошенную информацию.

Более подробно все показано на рис. 1.8, где приведены диаграммы классов, представляющие систему `Employee/Payroll`, о которой мы ведем речь.

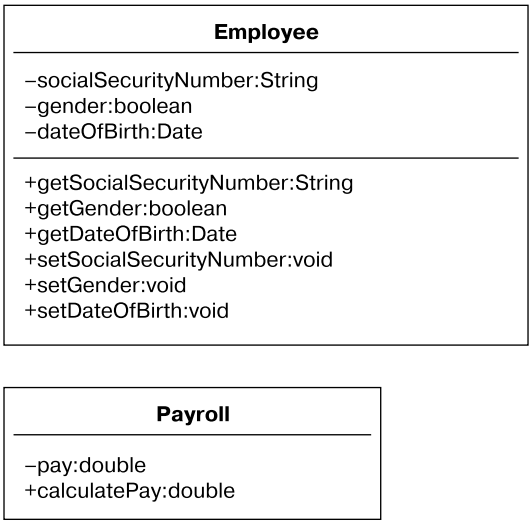

Рис. 1.8. Диаграммы классов Employee и Payroll

UML-ДИАГРАММЫ КЛАССОВ

Это были первые диаграммы классов, которые мы рассмотрели. Как видите, они весьма просты и лишены части конструкций (таких, например, как конструкторы), которые должен содержать надлежащий класс. Более подробно мы рассмотрим диаграммы классов и конструкторы в главе 3.

Каждая диаграмма определяется тремя отдельными секциями: именем как таковым, данными (атрибутами) и поведениями (методами). На рис. 1.8 показано, что секция атрибутов диаграммы класса `Employee` содержит `socialSecurityNumber`, `gender` и `dateofBirth`, в то время как секция методов включает методы, которые оперируют этими атрибутами. Вы можете использовать средства моделирования UML для создания и сопровождения диаграмм классов, соответствующих реальному коду.

СРЕДСТВА МОДЕЛИРОВАНИЯ

Средства визуального моделирования обеспечивают механизм для создания и манипулирования диаграммами классов с использованием унифицированного языка моделирования Unified Modeling Language (UML). Диаграммы классов расматриваются по ходу всей книги, и вы можете найти описание этой нотации в главе 10. UML-диаграммы классов используются как средство, помогающее визуализировать классы и их взаимоотношения с другими классами. Использование UML в этой книге ограничивается диаграммами классов.

О взаимоотношениях между классами и объектами мы поговорим позднее в этой главе, а пока вы можете представлять себе класс как шаблон, на основе которого создаются объекты. При создании объектов мы говорим, что создаются экземпляры этих объектов. Таким образом, если мы создадим три `Employee`, то на самом деле сгенерируем три полностью отдельных экземпляра класса `Employee`. Каждый объект

будет содержать собственную копию атрибутов и методов. Например, взгляните на рис. 1.9. Объект `Employee` с именем `John` (которое является его идентификатором) включает собственную копию всех атрибутов и методов, определенных в классе `Employee`. Объект `Employee` с именем `Mary` тоже содержит собственную копию атрибутов и методов. Оба объекта включают в себя отдельную копию атрибута `dateOfBirth` и метода `getDateOfBirth`.

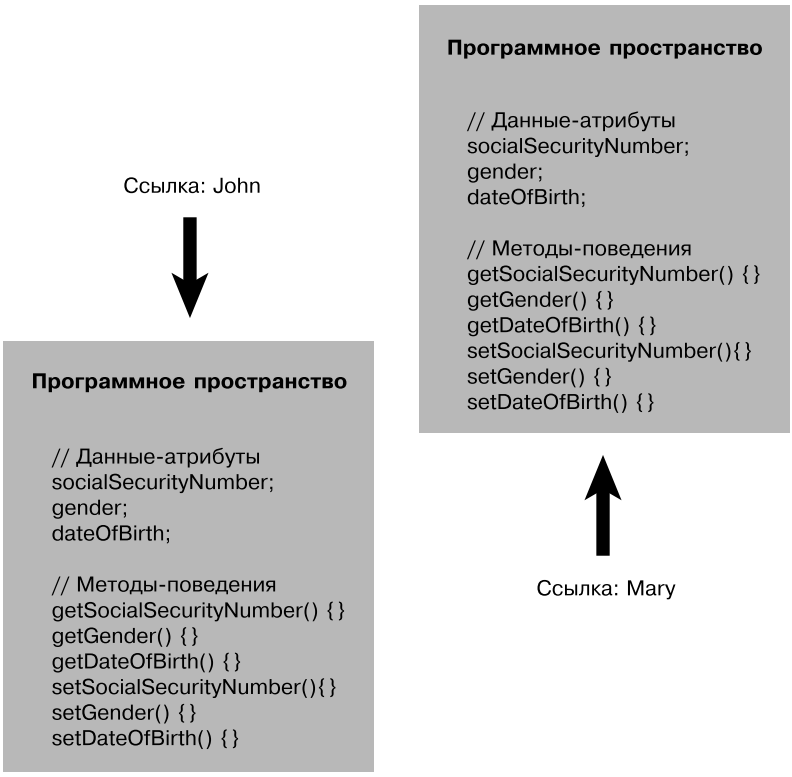

Рис. 1.9. Программные пространства

ВОПРОС РЕАЛИЗАЦИИ

Знайте, что не обязательно располагать физической копией каждого метода для каждого объекта. Лучше, чтобы каждый объект указывал на одну и ту же реализацию. Однако решение этого вопроса будет зависеть от используемого компилятора/операционной платформы. На концептуальном уровне вы можете представлять себе объекты как полностью независимые и содержащие собственные атрибуты и методы.

Что такое класс

Если говорить просто, то класс — это «чертеж» объекта. При создании экземпляра объекта вы станете использовать класс как основу для того, как этот объект будет

создаваться. Фактически попытка объяснить классы и объекты подобна стремлению решить дилемму «что было раньше — курица или яйцо?». Трудно описать класс без использования термина *«объект»* и наоборот. Например, велосипед определенного человека — это объект. Однако для того, чтобы построить этот велосипед, кому-то сначала пришлось подготовить чертежи (то есть класс), по которым он затем был изготовлен.

В случае с объектно-ориентированным программным обеспечением, в отличие от дилеммы «что было раньше — курица или яйцо?», мы знаем, что первым был именно класс. Нельзя создать экземпляр объекта без класса. Таким образом, многие концепции в этом разделе схожи с теми, что были представлены ранее в текущей главе, особенно если вести речь об атрибутах и методах.

Для объяснения классов и методов целесообразно использовать пример из сферы реляционных баз данных. Если говорить о таблице базы данных, то определением этой таблицы как таковой (полей, описания и использованных типов данных) был бы класс (метаданные), а объектами выступали бы строки таблицы (данные).

Эта книга сосредоточена на концепциях объектно-ориентированного программного обеспечения, а не на конкретной реализации (вроде Java, C#, Visual Basic .NET, Objective C или C++), однако зачастую полезно использовать примеры кода для объяснения некоторых концепций, поэтому фрагменты кода на Java задействуются по ходу всей этой книги, в соответствующих случаях помогая в объяснении отдельных тем. Однако, когда это целесообразно, в конце некоторых глав приводятся примеры использованного в них кода на C#.

В последующих разделах описываются некоторые фундаментальные концепции классов и то, как они взаимодействуют друг с другом.

Создание объектов

Классы можно представлять себе как шаблоны или формочки для печенья для объектов, как показано на рис. 1.10. Класс используется для создания объекта.

Класс можно представлять себе как нечто вроде типа данных более высокого уровня. Например, точно таким же путем, каким вы создаете то, что относится к типу данных `int` или `float`:

```
int x;
float y;
```

вы можете создать объект с использованием предопределенного класса:

```
myClass myObject;
```

В этом примере сами имена явно свидетельствуют о том, что `myClass` является классом, а `myObject` — объектом.

Помните, что каждый объект содержит собственные атрибуты (данные) и поведения (функции или программы). Класс определяет атрибуты и поведения, которые будут принадлежать всем объектам, созданным с использованием этого класса. Классы — это фрагменты кода. Объекты, экземпляры которых созданы на основе классов, можно распространять по отдельности либо как часть библиотеки. Объекты создаются на основе классов, поэтому классы должны определять базовые

строительные блоки объектов (атрибуты, поведения и сообщения). В общем, вам потребуется спроектировать класс прежде, чем вы сможете создать объект.

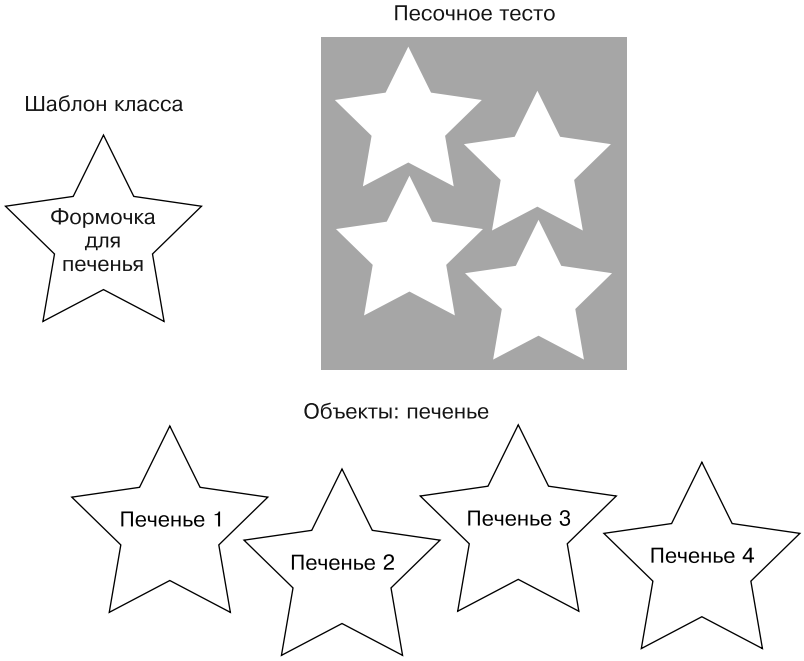

Рис. 1.10. Шаблон класса

Вот, к примеру, определение класса `Person`:

```
public class Person{

  // Атрибуты
  private String name;
  private String address;

  // Методы
  public String getName(){
    return name;
  }
  public void setName(String n){
    name = n;
  }

  public String getAddress(){
    return address;
  }
  public void setAddress(String adr){
    address = adr;
  }
}
```

Атрибуты

Как вы уже видели, данные класса представляются атрибутами. Любой класс должен определять атрибуты, сохраняющие состояние каждого объекта, экземпляр которого окажется создан на основе этого класса. Если рассматривать класс `Person` из предыдущего раздела, то он определяет атрибуты для `name` и `address`.

ОБОЗНАЧЕНИЯ ДОСТУПА

Когда тип данных или метод определен как public, у других объектов будет к нему прямой доступ. Когда тип данных или метод определен как private, только конкретный объект сможет получить к нему доступ. Еще один модификатор доступа — protected — разрешает доступ с использованием связанных объектов, но на эту тему мы поговорим в главе 3.

Методы

Как вы узнали ранее из этой главы, методы реализуют требуемое поведение класса. Каждый объект, экземпляр которого окажется создан на основе этого класса, будет содержать методы, определяемые этим же классом. Методы могут реализовывать поведения, вызываемые из других объектов (с помощью сообщений) либо обеспечивать основное, внутреннее поведение класса. Внутренние поведения — это закрытые методы, которые недоступны другим объектам. В классе `Person` поведениями являются `getName()`, `setName()`, `getAddress()` и `setAddress()`. Эти методы позволяют другим объектам инспектировать и изменять значения атрибутов соответствующего объекта. Это методика, широко распространенная в сфере объектно-ориентированных систем. Во всех случаях доступ к атрибутам в объекте должен контролироваться самим этим объектом — никакие другие объекты не должны напрямую изменять значения атрибутов этого объекта.

Сообщения

Сообщения — это механизм коммуникаций между объектами. Например, когда объект A вызывает метод объекта B, объект A отправляет сообщение объекту B. Ответ объекта B определяется его возвращаемым значением. Только открытые, а не закрытые методы объекта могут вызываться другим объектом. Приведенный далее код демонстрирует эту концепцию:

```
public class Payroll{

  String name;

  Person p = new Person();

  String = p.setName("Joe");

  ...код

  String = p.getName();

}
```

В этом примере (предполагая, что был создан экземпляр объекта `Payroll`) объект `Payroll` отправляет сообщение объекту `Person` с целью извлечения имени с помощью метода `getName()`. Опять-таки не стоит слишком беспокоиться о фактическом коде, поскольку в действительности нас интересуют концепции. Мы подробно рассмотрим код по мере нашего продвижения по этой книге.

Использование диаграмм классов в качестве визуального средства

С годами разрабатывается множество средств и методологий моделирования, призванных помочь в проектировании программных систем. Я с самого начала использовал UML-диаграммы классов как вспомогательный инструмент в образовательном процессе. Несмотря на то что подробное описание UML лежит вне рамок этой книги, мы будем использовать UML-диаграммы классов для иллюстрирования создаваемых классов. Фактически мы уже использовали диаграммы классов в этой главе. На рис. 1.11 показана диаграмма класса `Person`, о котором шла речь ранее в этой главе.

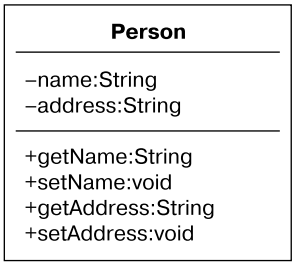

Рис. 1.11. Диаграмма класса Person

Обратите внимание, что атрибуты и методы разделены (атрибуты располагаются вверху, а методы — внизу). По мере того как мы будем сильнее углубляться в объектно-ориентированное проектирование, диаграммы классов будут становиться значительно сложнее и сообщать намного больше информации о том, как разные классы взаимодействуют друг с другом.

Инкапсуляция и скрытие данных

Одно из основных преимуществ использования объектов заключается в том, что объекту не нужно показывать все свои атрибуты и поведения. При хорошем объектно-ориентированном проектировании (по крайней мере, при таком, которое повсеместно считается хорошим) объект должен показывать только интерфейсы, необходимые другим объектам для взаимодействия с ним. Детали, не относящиеся к использованию объекта, должны быть скрыты от всех других объектов.

Инкапсуляция определяется тем, что объекты содержат как атрибуты, так и поведения. Скрытие данных является основной частью инкапсуляции.

Например, объект, который применяется для вычисления квадратов чисел, должен обеспечивать интерфейс для получения результатов. Однако внутренние атрибуты и алгоритмы, используемые для вычисления квадратов чисел, не нужно делать доступными для запрашивающего объекта. Надежные классы проектируются с учетом инкапсуляции. В последующих разделах мы рассмотрим концепции интерфейса и реализации, которые образуют основу инкапсуляции.

Интерфейсы

Мы уже видели, что интерфейс определяет основные средства коммуникации между объектами. При проектировании любого класса предусматриваются интерфейсы для надлежащего создания экземпляров и эксплуатации объектов. Любое поведение, которое обеспечивается объектом, должно вызываться через сообщение, отправляемое с использованием одного из предоставленных интерфейсов. В случае с интерфейсом должно предусматриваться полное описание того, как пользователи соответствующего класса будут взаимодействовать с этим классом. В большинстве объектно-ориентированных языков программирования методы, являющиеся частью интерфейсов, определяются как `public`.

ЗАКРЫТЫЕ ДАННЫЕ

Для того чтобы скрытие данных произошло, все атрибуты должны быть объявлены как private. Поэтому атрибуты никогда не являются частью интерфейсов. Частью интерфейсов классов могут быть только открытые методы. Объявление атрибута как public нарушает концепцию скрытия данных.

Взглянем на пример того, о чем совсем недавно шла речь: рассмотрим вычисление квадратов чисел. В таком примере интерфейс включал бы две составляющие:

- способ создать экземпляр объекта `Square`;
- способ отправить значение объекту и получить в ответ квадрат соответствующего числа.

Как уже отмечалось ранее в этой главе, если пользователю потребуется доступ к атрибуту, то будет сгенерирован метод для возврата значения этого атрибута (геттер). Если затем пользователю понадобится получить значение атрибута, то будет вызван метод для возврата его значения. Таким образом, объект, содержащий атрибут, будет управлять доступом к нему. Это жизненно важно, особенно в плане безопасности, тестирования и сопровождения. Если вы контролируете доступ к атрибуту, то при возникновении проблемы вам не придется беспокоиться об отслеживании каждого фрагмента кода, который мог бы изменить значение соответствующего атрибута — оно может быть изменено только в одном месте (с помощью сеттера).

С точки зрения безопасности вам не нужен неконтролируемый код для изменения или извлечения таких данных, как пароли и личная информация.

ПОДПИСИ: ИНТЕРФЕЙСЫ В СОПОСТАВЛЕНИИ С ИНТЕРФЕЙСАМИ

Важно отметить, что существуют интерфейсы как для классов, так и для методов, поэтому не путайте их. Интерфейсы классов — это открытые методы. Их вызов осуществляется при использовании их подписи, которая главным образом состоит из имени метода и списка его параметров. Более подробно эта концепция будет рассмотрена позднее.

Реализации

Только открытые атрибуты и методы являются частью интерфейсов. Пользователи не должны видеть какую-либо часть внутренней реализации и могут взаимодействовать с объектами исключительно через интерфейсы классов. Таким образом, все определенное как `public` окажется недоступно пользователям и будет считаться частью внутренней реализации классов.

В приводившемся ранее примере с классом `Employee` были скрыты только атрибуты. Во многих ситуациях будут попадаться методы, которые также должны быть скрыты и, таким образом, не являться частью интерфейса. В продолжение примера из предыдущего раздела представим, что речь идет о вычислении квадратного корня, и отметим при этом, что пользователям будет все равно, как вычисляется квадратный корень, при условии, что ответ окажется правильным. Таким образом, реализация может меняться, однако она не повлияет на пользовательский код. Например, компания, которая производит калькуляторы, может заменить алгоритм (возможно, потому, что новый алгоритм оказался более эффективным), что не повлияет при этом на результаты.

Реальный пример парадигмы «интерфейс/реализация»

На рис. 1.12 проиллюстрирована парадигма «интерфейс/реализация» с использованием реальных объектов, а не кода. Тостеру для работы требуется электричество. Чтобы обеспечить подачу электричества, нужно вставить вилку шнура тостера в электрическую розетку, которая является интерфейсом. Для того чтобы получить требуемое электричество, тостеру нужно лишь «реализовать» шнур, который соответствует техническим характеристикам электрической розетки; это и есть интерфейс между тостером и электроэнергетической компанией (в действительности — электроэнергетикой). Для тостера не имеет значения, что фактической реализацией является электростанция, работающая на угле. На самом деле для него важно лишь то, окажется реализацией атомная электростанция или же локальный электрогенератор. При такой модели любой электроприбор сможет получить электричество, если он соответствует спецификации интерфейса, как показано на рис. 1.12.

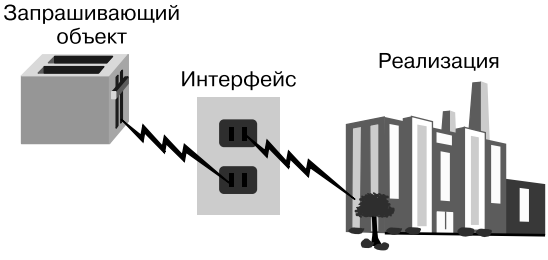

Рис. 1.12. Пример с электростанцией

Модель парадигмы «интерфейс/реализация»

Подробнее разберем класс `Square`. Допустим, вы создаете класс для вычисления квадратов целых чисел. Вам потребуется обеспечить отдельный интерфейс и реализацию. Иначе говоря, вы должны будете предусмотреть для пользователей способ вызова методов и получения квадратичных значений. Вам также потребуется обеспечить реализацию, которая вычисляет квадраты чисел; однако пользователям не следует что-либо знать о конкретной реализации. На рис. 1.13 показан один из способов сделать это. Обратите внимание, что на диаграмме класса знак плюса (+) обозначает `public`, а знак минуса (-) указывает на `private`. Таким образом, вы сможете идентифицировать интерфейс по методам, в начале которых стоит знак плюса.

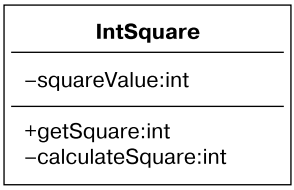

Рис. 1.13. Класс IntSquare

Эта диаграмма класса соответствует следующему коду:

```
public class IntSquare {

  // закрытый атрибут
  private int squareValue;

  // открытый интерфейс
  public int getSquare (int value) {

    SquareValue =calculateSquare(value);

    return squareValue;

  }

  // закрытая реализация
  private int calculateSquare (int value) {

    return value*value;
  }
}
```

Следует отметить, что единственной частью класса, доступной для пользователей, является открытый метод `getSquare`, который относится к интерфейсу. Реализация алгоритма вычисления квадратов чисел заключена в закрытом методе `calculateSquare`. Обратите также внимание на то, что атрибут `SquareValue` является

закрытым, поскольку пользователям не нужно знать о его наличии. Поэтому мы скрыли часть реализации: объект показывает только интерфейсы, необходимые пользователям для взаимодействия с ним, а детали, не относящиеся к использованию объекта, скрыты от других объектов.

Если бы потребовалось сменить реализацию — допустим, вы захотели бы использовать встроенную квадратичную функцию соответствующего языка программирования, — то вам не пришлось бы менять интерфейс. Вот код, использующий метод `Math.pow` из Java-библиотеки, который выполняет ту же функцию, однако обратите внимание, что `calculateSquare` по-прежнему является частью интерфейса:

```
// закрытая реализация
private int calculateSquare (int value) {

  return = Math.pow(value,2);

}
```

Пользователи получат ту же самую функциональность с применением того же самого интерфейса, однако реализация будет другой. Это очень важно при написании кода, который будет иметь дело с данными. Так, например, вы сможете перенести данные из файла в базу данных, не заставляя пользователя вносить изменения в какой-либо программный код.

Наследование

Одной из наиболее сильных сторон объектно-ориентированного программирования, пожалуй, является повторное использование кода. При структурном проектировании повторное использование кода допускается в известной мере: вы можете написать процедуру, а затем применять ее столько раз, сколько пожелаете. Однако объектно-ориентированное проектирование делает важный шаг вперед, позволяя вам определять отношения между классами, которые не только облегчают повторное использование кода, но и способствуют созданию лучшей общей конструкции путем упорядочения и учета общности разнообразных классов. Основное средство обеспечения такой функциональности — *наследование*.

Наследование позволяет классу наследовать атрибуты и методы другого класса. Это дает возможность создавать абсолютно новые классы путем абстрагирования общих атрибутов и поведений.

Одна из основных задач проектирования при объектно-ориентированном программировании заключается в выделении общности разнообразных классов. Допустим, у вас есть класс `Dog` и класс `Cat`, каждый из которых будет содержать атрибут `eyeColor`. При процедурной модели код как для `Dog`, так и для `Cat` включал бы этот атрибут. При объектно-ориентированном проектировании атрибут, связанный с цветом, можно перенести в класс с именем `Mammal` наряду со всеми прочими общими атрибутами и методами. В данном случае оба класса — `Dog` и `Cat` — будут наследовать от класса `Mammal`, как показано на рис. 1.14.

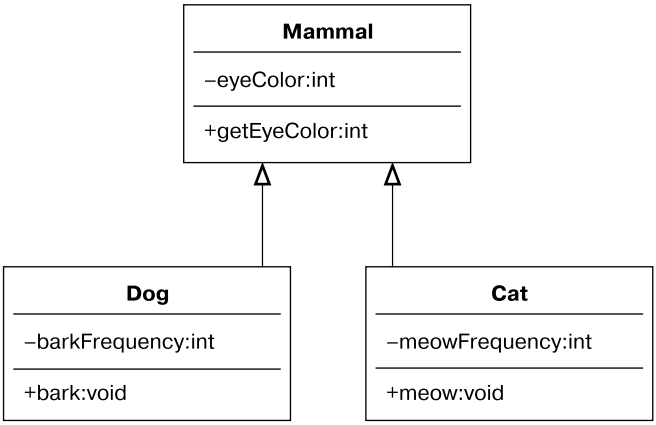

Рис. 1.14. Иерархия классов млекопитающих

Итак, оба класса наследуют от `Mammal`. Это означает, что в итоге класс `Dog` будет содержать следующие атрибуты:

```
eyeColor        // унаследован от Mammal
barkFrequency   // определен только для Dog
```

В том же духе объект `Dog` будет содержать следующие методы:

```
getEyeColor  // унаследован от Mammal
bark         // определен только для Dog
```

Создаваемый экземпляр объекта `Dog` или `Cat` будет содержать все, что есть в его собственном классе, а также все имеющееся в родительском классе. Таким образом, `Dog` будет включать все свойства своего определения класса, а также свойства, унаследованные от класса `Mammal`.

Суперклассы и подклассы

Суперкласс, или родительский класс (иногда называемый базовым), содержит все атрибуты и поведения, общие для классов, которые наследуют от него. Например, в случае с классом `Mammal` все классы млекопитающих содержат аналогичные атрибуты, такие как `eyeColor` и `hairColor`, а также поведения вроде `generateInternalHeat` и `growHair`. Все классы млекопитающих включают эти атрибуты и поведения, поэтому нет необходимости дублировать их, спускаясь по дереву наследования, для каждого типа млекопитающих. Дублирование потребует много дополнительной работы, и, пожалуй, что вызывает наибольшее беспокойство, оно может привести к ошибкам и несоответствиям.

Подкласс, или дочерний класс (иногда называемый производным), представляет собой расширение суперкласса. Таким образом, классы `Dog` и `Cat` наследуют все общие атрибуты и поведения от класса `Mammal`. Класс `Mammal` считается суперклассом подклассов, или дочерних классов, `Dog` и `Cat`.

Наследование обеспечивает большое количество преимуществ в плане проектирования. При проектировании класса `Cat` класс `Mammal` предоставляет значительную часть

требуемой функциональности. Наследуя от объекта `Mammal`, `Cat` уже содержит все атрибуты и поведения, которые делают его настоящим классом млекопитающих. Точнее говоря, являясь классом млекопитающих такого типа, как кошки, `Cat` должен включать любые атрибуты и поведения, которые свойственны исключительно кошкам.

Абстрагирование

Дерево наследования может разрастись довольно сильно. Когда классы `Mammal` и `Cat` будут готовы, добавить другие классы млекопитающих, например собак (или львов, тигров и медведей), не составит особого труда. Класс `Cat` также может выступать в роли суперкласса. Например, может потребоваться дополнительно абстрагировать `Cat`, чтобы обеспечить классы для персидских, сиамских кошек и т. д. Точно так же, как и `Cat`, класс `Dog` может выступать в роли родительского класса для других классов, например `GermanShepherd` и `Poodle` (рис. 1.15). Мощь наследования заключается в его методиках абстрагирования и организации.

В большинстве объектно-ориентированных языков программирования (например, Java, .NET и Objective C) у класса может иметься только один родительский, но много дочерних классов. А в некоторых языках программирования, например C++, у одного класса может быть несколько родительских классов. В первом случае наследование называется *простым*, а во втором — *множественным*.

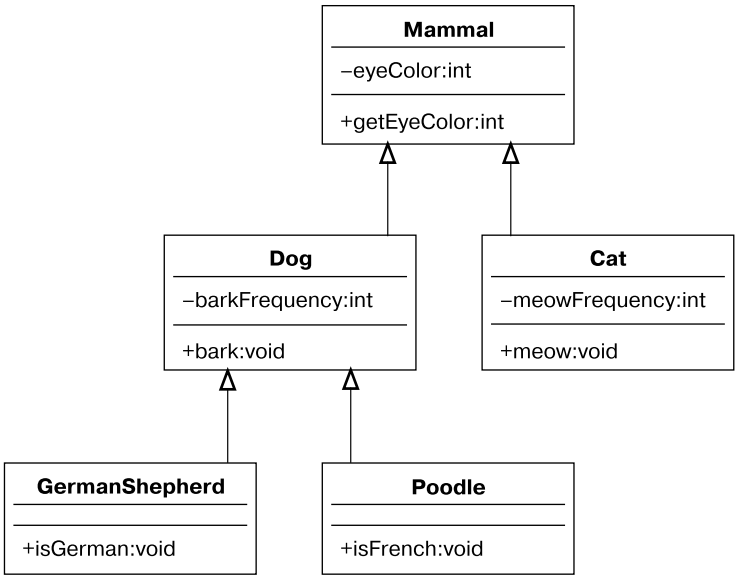

Рис. 1.15. UML-диаграмма классов млекопитающих

Обратите внимание, что оба класса — `GermanShepherd` и `Poodle` — наследуют от `Dog` — каждый содержит только один метод. Однако, поскольку они наследуют от `Dog`, они также наследуют от `Mammal`. Таким образом, классы `GermanShepherd` и `Poodle` включают в себя все атрибуты и методы, содержащиеся в `Dog` и `Mammal`, а также свои собственные (рис. 1.16).

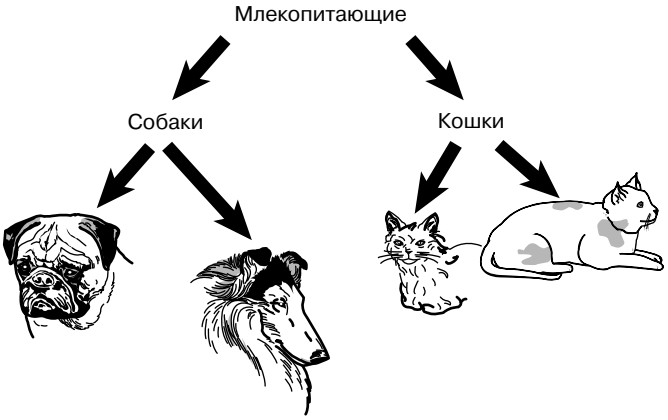

Рис. 1.16. Иерархия млекопитающих

Отношения «является экземпляром»

Рассмотрим пример, в котором `Circle`, `Square` и `Star` наследуют от `Shape`. Это отношение часто называется *отношением «является экземпляром»*, поскольку круг — это форма, как и квадрат. Когда подкласс наследует от суперкласса, он получает все возможности, которыми обладает этот суперкласс. Таким образом, `Circle`, `Square` и `Star` являются расширениями `Shape`.

На рис. 1.17 имя каждого из объектов представляет метод `Draw` для `Circle`, `Star` и `Square` соответственно. При проектировании системы `Shape` очень полезно было бы стандартизировать то, как мы используем разнообразные формы. Так мы могли бы решить, что, если нам потребуется нарисовать фигуру любой формы, мы вызовем метод с именем `Draw`. Если мы станем придерживаться этого решения всякий раз, когда нам нужно будет нарисовать фигуру, то потребуется вызывать только метод `Draw`, независимо от того, какой она будет формы. В этом заключается фундаментальная концепция полиморфизма — на индивидуальный объект, будь то `Circle`, `Star` или `Square`, возлагается обязанность по рисованию фигуры, которая ему соответствует. Это общая концепция во многих современных приложениях, например предназначенных для рисования и обработки текста.

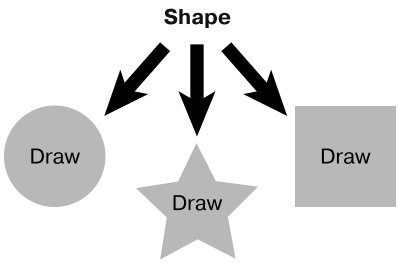

Рис. 1.17. Иерархия Shape

Полиморфизм

Полиморфизм — это греческое слово, буквально означающее множественность форм. Несмотря на то что полиморфизм тесно связан с наследованием, он часто упоминается отдельно от него как одно из наиболее весомых преимуществ объектно-ориентированных технологий. Если потребуется отправить сообщение объекту, он должен располагать методом, определенным для ответа на это сообщение. В иерархии наследования все подклассы наследуют от своих суперклассов. Однако, поскольку каждый подкласс представляет собой отдельную сущность, каждому из них может потребоваться дать отдельный ответ на одно и то же сообщение.

Возьмем, к примеру, класс `Shape` и поведение с именем `Draw`. Когда вы попросите кого-то нарисовать фигуру, первый вопрос вам будет звучать так: «Какой формы?» Никто не сможет нарисовать требуемую фигуру, не зная формы, которая является абстрактной концепцией (кстати, метод `Draw()` в коде `Shape` не содержит реализации). Вы должны указать конкретную форму. Для этого потребуется обеспечить фактическую реализацию в `Circle`. Несмотря на то что `Shape` содержит метод `Draw`, `Circle` переопределит этот метод и обеспечит собственный метод `Draw()`. Переопределение, в сущности, означает замену реализации родительского класса на реализацию из дочернего класса.

Допустим, у вас имеется массив из трех форм — `Circle`, `Square` и `Star`. Даже если вы будете рассматривать их все как объекты `Shape` и отправите сообщение `Draw` каждому объекту `Shape`, то конечный результат для каждого из них будет разным, поскольку `Circle`, `Square` и `Star` обеспечивают фактические реализации. Одним словом, каждый класс способен реагировать на один и тот же метод `Draw` не так, как другие, и рисовать соответствующую фигуру. Это и понимается под полиморфизмом.

Взгляните на следующий класс `Shape`:

```
public abstract class Shape{

  private double area;

  public abstract double getArea();
}
```

Класс `Shape` включает атрибут с именем `area`, который содержит значение площади фигуры. Метод `getArea()` включает идентификатор с именем `abstract`. Когда метод определяется как `abstract`, подкласс должен обеспечивать реализацию для этого метода; в данном случае `Shape` требует, чтобы подклассы обеспечивали реализацию `getArea()`. А теперь создадим класс с именем `Circle`, который будет наследовать от `Shape` (ключевое слово `extends` будет указывать на то, что `Circle` наследует от `Shape`):

```
public class Circle extends Shape{

  double radius;
```

```
    public Circle(double r) {

      radius = r;

    }

    public double getArea() {

      area = 3.14*(radius*radius);
      return (area);

    }
  }
```

Здесь мы познакомимся с новой концепцией под названием *«конструктор»*. Класс `Circle` содержит метод с таким же именем — `Circle`. Если имя метода оказывается аналогичным имени класса и при этом не предусматривается возвращаемого типа, то это особый метод, называемый конструктором. Считайте конструктор точкой входа для класса, где создается объект. Конструктор хорошо подходит для выполнения инициализаций и задач, связанных с запуском.

Конструктор `Circle` принимает один параметр, представляющий радиус, и присваивает его атрибуту `radius` класса `Circle`.

Класс `Circle` также обеспечивает реализацию для метода `getArea`, изначально определенного как `abstract` в классе `Shape`.

Мы можем создать похожий класс с именем `Rectangle`:

```
public class Rectangle extends Shape{
  double length;
  double width;

  public Rectangle(double l, double w){
    length = l;
    width = w;
  }

  public double getArea() {
    area = length*width;
    return (area);
  }

}
```

Теперь мы можем создавать любое количество классов прямоугольников, кругов и т. д. и вызывать их метод `getArea()`. Ведь мы знаем, что все классы прямоугольников и кругов наследуют от `Shape`, а все классы `Shape` содержат метод `getArea()`. Если подкласс наследует абстрактный метод от суперкласса, то он должен обеспечивать конкретную реализацию этого метода, поскольку иначе он сам будет абстрактным классом (см. рис. 1.18, где приведена UML-диаграмма). Этот подход также обеспечивает механизм для довольно легкого создания других, новых классов.

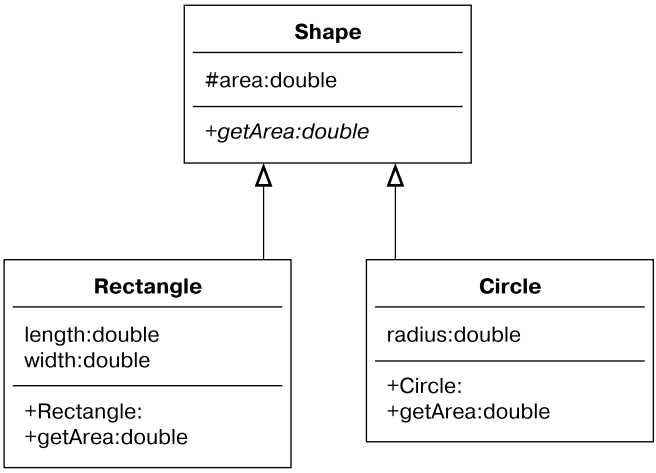

Рис. 1.18. UML-диаграмма Shape

Таким образом, мы можем создать экземпляры классов Shape следующим путем:

```
Circle circle = new Circle(5);
Rectangle rectangle = new Rectangle(4,5);
```

Затем, используя такую конструкцию, как стек, мы можем добавить в него классы Shape:

```
stack.push(circle);
stack.push(rectangle);
```

ЧТО ТАКОЕ СТЕК?

Стек — это структура данных, представляющая собой систему «последним поступил — первым ушел». Это как стопка монет в форме цилиндра, которые вы складываете одна на другую. Когда вам потребуется монета, вы снимете верхнюю монету, которая при этом окажется последней, которую вы положили в стопку. Вставка элемента в стек означает, что вы добавляете его на вершину стека (подобно тому как вы кладете следующую монету в стопку). Удаление элемента из стека означает, что вы убираете последний элемент из стека (подобно снятию верхней монеты).

Теперь переходим к увлекательной части. Мы можем очистить стек, и нам при этом не придется беспокоиться о том, какие классы Shape в нем находятся (мы просто будем знать, что они связаны с фигурами):

```
while ( !stack.empty()) {
  Shape shape = (Shape) stack.pop();
  System.out.println ("Площадь = " + shape.getArea());
}
```

В действительности мы отправляем одно и то же сообщение всем Shape:

```
shape.getArea()
```

Однако фактическое поведение, которое имеет место, зависит от типа фигуры. Например, `Circle` вычисляет площадь круга, а `Rectangle` — площадь прямоугольника. На самом деле (и в этом состоит ключевая концепция) мы отправляем сообщение классам `Shape` и наблюдаем разное поведение в зависимости от того, какие подклассы `Shape` используются.

Этот подход направлен на обеспечение стандартизации определенного интерфейса среди классов, а также приложений. Представьте себе приложение из офисного пакета, которое позволяет обрабатывать текст, и приложение для работы с электронными таблицами. Предположим, что они оба включают класс с именем `Office`, который содержит интерфейс с именем `print()`. Этот `print()` необходим всем классам, являющимся частью офисного пакета. Любопытно, но несмотря на то, что текстовый процессор и табличная программа вызывают интерфейс `print()`, они делают разные вещи: один выводит текстовый документ, а другая — документ с электронными таблицами.

Композиция

Вполне естественно представлять себе, что одни объекты содержат другие объекты. У телевизора есть тюнер и экран. У компьютера есть видеокарта, клавиатура и жесткий диск. Хотя компьютер сам по себе можно считать объектом, его жесткий диск тоже считается полноценным объектом.

Фактически вы могли бы открыть системный блок компьютера, достать жесткий диск и подержать его в руке. Как компьютер, так и его жесткий диск считаются объектами. Просто компьютер содержит другие объекты, например жесткий диск.

Таким образом, объекты зачастую формируются или состоят из других объектов — это и есть композиция.

Абстрагирование

Точно так же, как и наследование, композиция обеспечивает механизм для создания объектов. Фактически я сказал бы, что есть только два способа создания классов из других классов: *наследование* и *композиция*. Как мы уже видели, наследование позволяет одному классу наследовать от другого. Поэтому мы можем абстрагировать атрибуты и поведения для общих классов. Например, как собаки, так и кошки относятся к млекопитающим, поскольку собака *является экземпляром* млекопитающего так же, как и кошка. Благодаря композиции мы к тому же можем создавать классы, вкладывая одни классы в другие.

Взглянем на отношение между автомобилем и двигателем. Преимущества разделения двигателя и автомобиля очевидны. Создавая двигатель отдельно, мы сможем использовать его в разных автомобилях — не говоря уже о других преимуществах. Однако мы не можем сказать, что двигатель *является экземпляром* автомобиля. Это будет просто неправильно звучать, если так выразиться (а поскольку мы моделируем реальные системы, это нам и нужно). Вместо этого для описания отношений композиции мы используем словосочетание *«содержит как часть»*. Автомобиль *содержит как часть* двигатель.

Отношения «содержит как часть»

Хотя отношения наследования считаются *отношениями «является экземпляром»* по тем причинам, о которых мы уже говорили ранее, отношения композиции называются *отношениями «содержит как часть»*. Если взять пример из приводившегося ранее раздела, то телевизор *содержит как часть* тюнер, а также экран. Телевизор, несомненно, не является тюнером, поэтому здесь нет никаких отношений наследования. В том же духе *частью* компьютера является видеокарта, клавиатура и жесткий диск. Тема наследования, композиции и того, как они соотносятся друг с другом, очень подробно разбирается в главе 7.

Резюме

При рассмотрении объектно-ориентированных технологий нужно много чего охватить. Однако по завершении чтения этой главы у вас должно сложиться хорошее понимание следующих концепций.

- **Инкапсуляция.** Инкапсуляция данных и поведений в одном объекте имеет первостепенное значение в объектно-ориентированной разработке. Один объект будет содержать как свои данные, так и поведения, и сможет скрыть то, что ему потребуется, от других объектов.
- **Наследование.** Класс может наследовать от другого класса и использовать преимущества атрибутов и методов, определяемых суперклассом.
- **Полиморфизм.** Означает, что схожие объекты способны по-разному отвечать на одно и то же сообщение. Например, у вас может быть система с множеством фигур.

  Однако круг, квадрат и звезда рисуются по-разному. Используя полиморфизм, вы можете отправить одно и то же сообщение (например, `Draw`) объектам, на каждый из которых возлагается обязанность по рисованию соответствующей ему фигуры.
- **Композиция.** Означает, что объект формируется из других объектов.

В этой главе рассмотрены фундаментальные объектно-ориентированные концепции, в которых к настоящему времени вы уже должны хорошо разбираться.

Примеры кода, использованного в этой главе

Приведенный далее код написан на C# .NET. Эти примеры соответствуют Java-коду, продемонстрированному в текущей главе.

Пример TestPerson: C# .NET

```
using System;

namespace ConsoleApplication1
{
  class TestPerson
```

```
  {
    static void Main(string[] args)
    {

      Person joe = new Person();

      joe.Name = "joe";

      Console.WriteLine(joe.Name);

      Console.ReadLine();
    }
  }

  public class Person
  {

    // Атрибуты
    private String strName;
    private String strAddress;

    // Методы
    public String Name
    {
      get { return strName; }
      set { strName = value; }
    }

    public String Address
    {
      get { return strAddress; }
      set { strAddress = value; }
    }
  }
}
```

Пример TestShape: C# .NET

```
using System;

namespace TestShape
{
  class TestShape
  {
    public static void Main()
    {

      Circle circle = new Circle(5);
      Console.WriteLine(circle.calcArea());

      Rectangle rectangle = new Rectangle(4, 5);
```

```
        Console.WriteLine(rectangle.calcArea());

        Console.ReadLine();

      }
    }
    public abstract class Shape
    {

      protected double area;

      public abstract double calcArea();

    }
    public class Circle : Shape
    {
      private double radius;

      public Circle(double r)
      {

        radius = r;

      }
      public override double calcArea()
      {

        area = 3.14 * (radius * radius);
        return (area);

      }
    }
    public class Rectangle : Shape
    {

      private double length;
      private double width;

      public Rectangle(double l, double w)
      {
        length = l;
        width = w;
      }

      public override double calcArea()
      {
        area = length * width;
        return (area);
      }
    }
  }
```

Глава 2
Как мыслить объектно

В главе 1 вы изучили фундаментальные объектно-ориентированные концепции. В остальной части этой книги мы тщательнее разберем эти концепции и познакомимся с некоторыми другими. Для грамотного подхода к проектированию необходимо учитывать много факторов, независимо от того, идет ли речь об объектно-ориентированном проектировании. В качестве основной единицы при объектно-ориентированном проектировании выступает класс. Желаемым конечным результатом такого проектирования является надежная и функциональная объектная модель, другими словами, полная система.

Как и в случае с большинством вещей в жизни, нет какого-то одного правильного или ошибочного подхода к устранению проблем. Обычно бывает много путей решения одной и той же проблемы. Поэтому, пытаясь выработать объектно-ориентированное решение, не зацикливайтесь на том, чтобы постараться с первого раза все идеально спроектировать (кое-что всегда можно будет усовершенствовать). В действительности вам потребуется прибегнуть к мозговому штурму и позволить мыслительному процессу пойти в разных направлениях. Не старайтесь соответствовать каким-либо стандартам или соглашениям, пытаясь решить проблему, поскольку важно лишь быть креативным.

Фактически на самом старте процесса не стоит даже начинать задумываться о конкретном языке программирования. Первым пунктом повестки дня должно быть определение и решение бизнес-проблем. Займитесь сперва концептуальным анализом и проектированием. Задумывайтесь о конкретных технологиях, только если они будут существенны для решения бизнес-проблем. Например, нельзя спроектировать беспроводную сеть без беспроводной технологии. Однако часто будет случаться так, что вам придется обдумывать сразу несколько программных решений.

Таким образом, перед тем как приступать к проектированию системы или даже класса, вам следует поразмыслить над соответствующей задачей и повеселиться! В этой главе мы рассмотрим изящное искусство и науку объектно-ориентированного мышления.

Любое фундаментальное изменение в мышлении не является тривиальным. Например, ранее много говорилось о переходе со структурной разработки на объектно-ориентированную. Один из побочных эффектов ведущихся при этом дебатов заключается в ошибочном представлении, что структурная и объектно-ориентированная разработки являются взаимоисключающими. Однако это не так. Как мы уже знаем из нашего исследования оберток, структурная и объектно-ориентированная разработки сосуществуют. Фактически, создавая объектно-ориентированное приложение,

вы повсеместно используете структурные конструкции. Мне никогда не доводилось видеть программу, объектно-ориентированную или любую другую, которая не задействует циклы, операторы `if` и т. д. Кроме того, переход на объектно-ориентированное проектирование не потребует каких-либо затрат.

Чтобы перейти с FORTRAN на COBOL или даже C, вам потребуется изучить новый язык программирования, однако для перехода с COBOL на C++, C# .NET, Visual Basic .NET, Objective-C или Java вам придется освоить новое мышление. Здесь всплывает избитое выражение *«объектно-ориентированная парадигма»*. При переходе на объектно-ориентированный язык вам сначала потребуется потратить время на изучение объектно-ориентированных концепций и освоение соответствующего мышления. Если такая смена парадигмы не произойдет, то случится одна из двух вещей: либо проект не окажется по-настоящему объектно-ориентированным по своей природе (например, он будет задействовать C++ без использования объектно-ориентированных конструкций), либо он окажется полной объектно-неориентированной неразберихой.

В этой главе рассматриваются три важные вещи, которые вы можете сделать для того, чтобы хорошо освоить объектно-ориентированное мышление:

- знать разницу между интерфейсом и реализацией;
- мыслить более абстрактно;
- обеспечивать для пользователей минимальный интерфейс из возможных.

Мы уже затронули некоторые из этих концепций в главе 1, а теперь разберемся в них более подробно.

Разница между интерфейсом и реализацией

Как мы уже видели в главе 1, один из ключей к грамотному проектированию — понимание разницы между интерфейсом и реализацией. Таким образом, при проектировании класса важно определить, что пользователю требуется знать, а что — нет. Механизм скрытия данных, присущий инкапсуляции, представляет собой инструмент, позволяющий скрывать от пользователей несущественные данные.

ПРЕДОСТЕРЕЖЕНИЕ

Не путайте концепцию интерфейса с терминами вроде «графический интерфейс пользователя» (GUI — Graphical User Interface). Несмотря на то что графический интерфейс пользователя, как видно из его названия, представляет собой интерфейс, используемый здесь термин является более общим по своей природе и не ограничивается понятием графического интерфейса.

Помните пример с тостером из главы 1? Тостер или любой электроприбор, если на то пошло, подключается к интерфейсу, которым является электрическая розет-

ка (рис. 2.1). Все электроприборы получают доступ к необходимому электричеству через электрическую розетку, которая соответствует нужному интерфейсу. Тостеру не нужно что-либо знать о реализации или о том, как вырабатывается электричество. Для него важно лишь то, чтобы работающая на угле или атомная электростанция могла вырабатывать электричество, — этому электроприбору все равно, какая из станций будет делать это, при условии, что интерфейс работает соответствующим образом, то есть корректно и надежно.

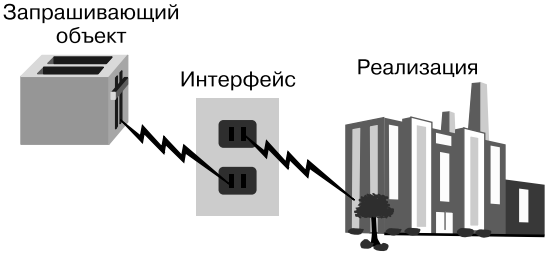

Рис. 2.1. Повторно приведенный пример с электростанцией

В качестве другого примера рассмотрим автомобиль. Интерфейс между вами и автомобилем включает такие компоненты, как руль, педаль газа, педаль тормоза и переключатель зажигания. Когда речь идет об управлении автомобилем, для большинства людей, если отбросить вопросы эстетики, главным является то, как он заводится, разгоняется, останавливается и т. д. Реализация, чем по сути является то, чего вы не видите, мало интересует среднестатистического водителя. Фактически большинство людей даже не способно идентифицировать определенные компоненты, например каталитический преобразователь или сальник. Однако любой водитель узнает руль и будет в курсе, как его использовать, поскольку это общий интерфейс. Устанавливая стандартный руль в автомобилях, производители могут быть уверены в том, что люди из их потенциального круга покупателей смогут использовать выпускаемую ими продукцию.

Однако если какой-нибудь производитель решит установить вместо руля джойстик, то большинство водителей будут разочарованы, а продажи таких автомобилей могут оказаться низкими (подобная замена устроит разве что отдельных эклектиков, которым нравится «двигаться против течения»). С другой стороны, если мощность и эстетика не изменятся, то среднестатистический водитель ничего не заметит, даже если производитель изменит двигатель (часть реализации) выпускаемых автомобилей.

Двигатель является частью реализации, а руль — частью интерфейса. Изменения в реализации не должны оказывать влияния на водителя, в то время как изменения в интерфейсе могут это делать. Водитель заметил бы эстетические изменения руля, даже если бы тот функционировал так же, как и раньше. Необходимо подчеркнуть, что изменения в двигателе, *заметные* для водителя, нарушают это правило. Например, изменение, которое приведет к заметной потере мощности, в действительности будет изменением интерфейса.

ЧТО ВИДЯТ ПОЛЬЗОВАТЕЛИ

Интерфейсы также непосредственно связаны с классами. Конечные пользователи обычно не видят каких-либо классов — они видят графический интерфейс пользователя или командную строку. Однако программисты увидят интерфейсы классов. Повторное использование классов означает, что эти классы кто-то уже написал. Поэтому программист, применяющий тот или иной класс, должен знать, как заставить его работать надлежащим образом. Этот программист будет комбинировать много классов для создания системы и должен разбираться в интерфейсах классов. Поэтому, когда в данной главе речь идет о пользователях, под ними подразумеваются проектировщики и разработчики, а не обязательно конечные пользователи. Таким образом, когда мы говорим об интерфейсах в этом контексте, речь идет об интерфейсах классов, а не о графических интерфейсах пользователей.

Должным образом сконструированные классы состоят из двух частей — интерфейса и реализации.

Интерфейс

Услуги, предоставляемые конечным пользователям, образуют интерфейс. В наиболее благоприятном случае конечным пользователям предоставляются *только* те услуги, которые им необходимы. Разумеется, то, какие услуги требуются определенному конечному пользователю, может оказаться спорным вопросом. Если вы поместите десять человек в одну комнату и попросите каждого из них спроектировать что-то независимо от других, то получите десять абсолютно разных результатов проектирования — и в этом не будет ничего плохого. Однако, как правило, интерфейс класса должен содержать то, что нужно знать пользователям. Если говорить о примере с тостером, то им необходимо знать только то, как подключить прибор к интерфейсу (которым в данном случае является электрическая розетка) и как эксплуатировать его.

ОПРЕДЕЛЕНИЕ ПОЛЬЗОВАТЕЛЕЙ

Пожалуй, наиболее важный момент при проектировании класса — определение его аудитории, или пользователей.

Реализация

Детали реализации скрыты от пользователей. Один из аспектов, касающихся реализации, которые нужно помнить, заключается в следующем: изменения в реализации *не должны* требовать внесения изменений в пользовательский код. В какой-то мере это может показаться сложным, однако в выполнении такого условия заключается суть соответствующей задачи проектирования. Если интерфейс спроектирован надлежащим образом, то изменения в реализации не должны требовать внесения изменений в пользовательский код. Помните, что интерфейс включает синтаксис для вызова методов и возврата значений. Если интерфейс не претерпит изменений, то пользователям будет все равно, изменится ли реализация. Важно лишь то, чтобы программисты смогли использовать аналогичный синтаксис и извлечь аналогичное значение.

Мы сталкиваемся с этим постоянно, когда пользуемся сотовыми телефонами. Интерфейс, применяемый для звонка, прост: мы либо набираем номер, либо выбираем тот, что имеется в адресной книге. Кроме того, если оператор связи обновит программное обеспечение, это не изменит способа, которым вы совершаете звонки. Интерфейс останется прежним независимо от того, как изменится реализация. Однако я могу представить себе ситуацию, что оператор связи изменил интерфейс: это произойдет, если изменится код города. При изменении основного интерфейса, как и кода города, пользователям придется действовать уже по-другому. Бизнес старается сводить к минимуму изменения такого рода, поскольку некоторым клиентам они не понравятся или, возможно, эти люди не захотят мириться с трудностями.

Напомню пример с тостером: хотя интерфейсом всегда является электрическая розетка, реализация может измениться с работающей на угле электростанции на атомную, никак не повлияв на тостер. Здесь следует сделать одну важную оговорку: работающая на угле или атомная электростанция тоже должна соответствовать спецификации интерфейса. Если работающая на угле электростанция обеспечивает питание переменного тока, а атомная — питание постоянного тока, то возникнет проблема. Основной момент здесь заключается в том, что и потребляющее электроэнергию устройство, и реализация должны соответствовать спецификации интерфейса.

Пример интерфейса/реализации

Создадим простой (пусть и не очень функциональный) класс `DataBaseReader`. Мы напишем Java-код, который будет извлекать записи из базы данных. Как уже говорилось ранее, знание своих конечных пользователей всегда является наиболее важным аспектом при проектировании любого рода. Вам следует провести анализ ситуации и побеседовать с конечными пользователями, а затем составить список требований к проекту. Далее приведены требования, которые нам придется учитывать при создании класса `DataBaseReader`. У нас должна быть возможность:

- открывать соединение с базой данных;
- закрывать соединение с базой данных;
- устанавливать указатель над первой записью в базе данных;
- устанавливать указатель над последней записью в базе данных;
- узнавать количество записей в базе данных;
- определять, есть ли еще записи в базе данных (если мы достигнем конца);
- устанавливать указатель над определенной записью путем обеспечения ключа;
- извлекать ту или иную запись путем обеспечения ключа;
- извлекать следующую запись исходя из позиции указателя.

Учитывая все эти требования, мы можем предпринять первую попытку спроектировать класс `DataBaseReader`, написав возможные интерфейсы для наших конечных пользователей.

В данном случае класс `DataBaseReader` предназначается для программистов, которым требуется использовать ту или иную базу данных. Таким образом,

интерфейс, в сущности, будет представлять собой интерфейс программирования приложений (API — Application Programming Interface), который станут использовать программисты. Соответствующие методы в действительности будут обертками, в которых окажется заключена функциональность, обеспечиваемая системой баз данных. Зачем нам все это нужно? Мы намного подробнее исследуем этот вопрос позднее в текущей главе, а короткий ответ звучит так: нам необходимо сконфигурировать кое-какую функциональность, связанную с базой данных. Например, нам может потребоваться обработать объекты для того, чтобы мы смогли записать их в реляционную базу данных. Создание соответствующего *промежуточного программного обеспечения* — непростая задача, поскольку предполагает проектирование и написание кода, однако представляет собой реальный пример обертывания функциональности. Более важно то, что нам может потребоваться изменить само ядро базы данных, но при этом не придется вносить изменения в код.

На рис. 2.2 показана диаграмма класса, представляющая возможный интерфейс для класса `DataBaseReader`.

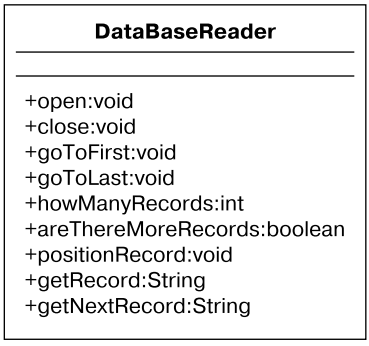

Рис. 2.2. UML-диаграмма класса DataBaseReader

Обратите внимание, что методы в этом классе открытые (помните, что рядом с именами методов, являющихся открытыми интерфейсами, присутствуют знаки плюса). Следует также отметить, что представлен только интерфейс, а реализация не показана. Уделите минуту на то, чтобы выяснить, отвечает ли в целом эта диаграмма класса требованиям к проекту, намеченным ранее. Если впоследствии вы обнаружите, что эта диаграмма отвечает не всем требованиям, то в этом не будет ничего плохого; помните, что объектно-ориентированное проектирование — это итеративный процесс, поэтому вам не обязательно стараться сделать все именно с первой попытки.

ОТКРЫТЫЙ ИНТЕРФЕЙС

Помните, что если метод является открытым, то прикладные программисты смогут получить к нему доступ и, таким образом, он будет считаться частью интерфейса класса. Не путайте термин «интерфейс» с ключевым словом interface, используемым в Java и .NET. На эту тему мы поговорим в последующих главах.

Для каждого из приведенных ранее требований нам необходим метод, обеспечивающий желаемую функциональность. Теперь нам нужно задать несколько вопросов.

- Чтобы эффективно использовать этот класс, нужно ли вам, как программисту, еще что-нибудь знать о нем?
- Нужно ли вам знать о том, как внутренний код базы данных открывает ее?
- Требуется ли вам знать о том, как внутренний код базы данных физически выбирает определенную запись?
- Нужно ли вам знать о том, как внутренний код базы определяет то, остались ли еще записи?

Ответом на все эти вопросы будет звучное *«нет»*! Вам не нужно знать что-либо из этой информации. Вам важно лишь получить соответствующие возвращаемые значения, а также то, что операции выполняются корректно. Кроме того, прикладные программисты, скорее всего, «будут на отдалении» как минимум еще одного абстрактного уровня от реализации. Приложение воспользуется вашими классами для открытия базы данных, что, в свою очередь, приведет к вызову соответствующего API-интерфейса для доступа к этой базе данных.

МИНИМАЛЬНЫЙ ИНТЕРФЕЙС

Один из способов, пусть даже экстремальных, определить минимальный интерфейс заключается в том, чтобы изначально не предоставлять пользователю никаких открытых интерфейсов. Разумеется, соответствующий класс будет бесполезным; однако это заставит пользователя вернуться к вам и сказать: «Эй, мне нужна эта функциональность». Тогда вы сможете начать переговоры. Таким образом, у вас есть возможность добавлять интерфейсы, только когда они запрашиваются. Никогда не предполагайте, что определенному пользователю что-то требуется.

Может показаться, что создание оберток — это перебор, однако их написание несет в себе множество преимуществ. Например, на рынке сегодня присутствует большое количество промежуточных продуктов. Рассмотрим проблему отображения объектов в реляционную базу данных. Некоторые объектно-ориентированные базы данных могут идеально подходить для объектно-ориентированных приложений. Однако есть одна маленькая проблема: у большинства компаний имеется множество данных в унаследованных системах управления реляционными базами. Как та или иная компания может использовать объектно-ориентированные технологии и быть передовой, сохраняя при этом свою информацию в реляционной базе данных?

Во-первых, вы можете преобразовать все свои унаследованные реляционные базы данных в совершенно новые объектно-ориентированные базы данных. Однако любому, кто испытывает острую боль от преобразования каких-либо данных, известно, что этого следует избегать любой ценой. Хотя на такие преобразования может уйти много времени и сил, зачастую они вообще не приводят к требуемым результатам.

Во-вторых, вы можете воспользоваться промежуточным продуктом для того, чтобы без проблем отобразить объекты, содержащиеся в коде вашего приложения,

в реляционную модель. Это более грамотное решение. Некоторые могут утверждать, что объектно-ориентированные базы данных намного эффективнее подходят для обеспечения постоянства объектов, чем реляционные базы данных. Фактически многие системы разработки без проблем обеспечивают такую функциональность.

ПОСТОЯНСТВО ОБЪЕКТОВ

Постоянство объектов относится к концепции сохранения состояния того или иного объекта для того, чтобы его можно было восстановить и использовать позднее. Объект, не являющийся постоянным, по сути «умирает», когда оказывается вне области видимости. Состояние объекта, к примеру, можно сохранить в базе данных.

В современной бизнес-среде отображение из реляционных баз данных в объектно-ориентированные — отличное решение. Многие компании интегрировали эти технологии. Компании используют распространенный подход, при котором внешний интерфейс сайта вместе с данными располагается на мейнфрейме.

Если вы создаете полностью объектно-ориентированную систему, то практичным (и более производительным) вариантом может оказаться объектно-ориентированная база данных. Вместе с тем объектно-ориентированные базы данных даже близко нельзя назвать такими же распространенными, как объектно-ориентированные языки программирования.

АВТОНОМНОЕ ПРИЛОЖЕНИЕ

Даже при создании нового объектно-ориентированного приложения с нуля может оказаться нелегко избежать унаследованных данных. Даже новое созданное объектно-ориентированное приложение, скорее всего, не будет автономным, и ему, возможно, потребуется обмениваться информацией, располагающейся в реляционных базах данных (или, собственно говоря, на любом другом устройстве накопления данных).

Вернемся к примеру с базой данных. На рис. 2.2 показан открытый интерфейс, который касается соответствующего класса, и ничего больше. Когда этот класс будет готов, в нем, вероятно, окажется больше методов, при этом он, несомненно, будет включать атрибуты. Однако вам, как программисту, который будет использовать этот класс, не потребуется что-либо знать о его закрытых методах и атрибутах. Вам, безусловно, не нужно знать, как выглядит код внутри его открытых методов. Вам просто понадобится быть в курсе, как взаимодействовать с интерфейсами.

Как выглядел бы код этого открытого интерфейса (допустим, мы начнем с примера базы данных Oracle)? Взглянем на метод `open()`:

```
public void open(String Name){

  /* Некая специфичная для приложения обработка */

  /* Вызов API-интерфейса Oracle для открытия базы данных */

  /* Еще некая специфичная для приложения обработка */

};
```

В данной ситуации вы, выступая в роли программиста, понимаете, что методу `open` в качестве параметра требуется `String`. `Name`, что представляет файл базы данных, передается, однако пояснение того, как `Name` отображается в определенную базу данных в случае с этим примером, не является важным. Это все, что нам нужно знать. А теперь переходим к увлекательному — что в действительности делает интерфейсы такими замечательными!

Чтобы досадить нашим пользователям, изменим реализацию базы данных. Представим, что вчера вечером мы преобразовали всю информацию из базы Oracle в информацию базы SQL Anywhere (при этом нам пришлось вынести острую боль). Эта операция заняла у нас несколько часов, но мы справились с ней.

Теперь код выглядит так:

```
public void open(String Name){

  /* Некая специфичная для приложения обработка */

  /* Вызов API-интерфейса SQL Anywhere для открытия базы данных */

  /* Еще некая специфичная для приложения обработка */

};
```

К нашему великому разочарованию, этим утром не поступило ни одной жалобы от пользователей. Причина заключается в том, что, хотя реализация изменилась, интерфейс не претерпел изменений! Что касается пользователей, то совершаемые ими вызовы остались такими же, как и раньше. Изменение кода реализации могло бы потребовать довольно много сил (а модуль с однострочным изменением кода пришлось бы перестраивать), однако не понадобилось бы изменять ни одной строки кода приложения, который задействует класс `DataBaseReader`.

ПЕРЕКОМПИЛЯЦИЯ КОДА

Динамически загружаемые классы загружаются во время выполнения, а не являются статически связанными с исполняемым файлом. При использовании динамически загружаемых классов, как, например, в случае с Java и .NET, не придется перекомпилировать ни один из пользовательских классов. Однако в языках программирования со статическим связыванием, например C++, для добавления нового класса потребуется связь.

Разделяя интерфейс пользователя и реализацию, мы сможем избежать головной боли в будущем. На рис. 2.3 реализации баз данных прозрачны для конечных пользователей, видящих только интерфейс.

Использование абстрактного мышления при проектировании классов

Одно из основных преимуществ объектно-ориентированного программирования состоит в том, что классы можно использовать повторно. Пригодные для повторного

применения классы обычно располагают интерфейсами, которые больше абстрактны, нежели конкретны. Конкретные интерфейсы склонны быть весьма специфичными, в то время как абстрактные являются более общими. Однако не всегда можно утверждать, что очень абстрактный интерфейс более полезен, чем очень конкретный, пусть это часто и является верным.

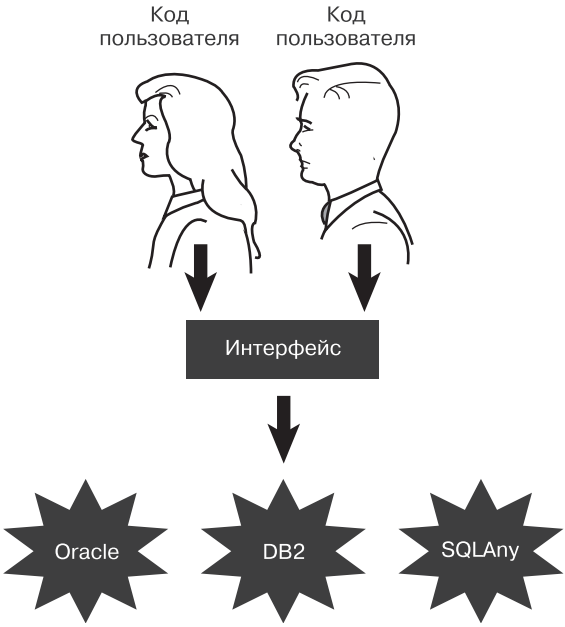

Рис. 2.3. Интерфейс

Можно создать очень полезный конкретный класс, который окажется вообще непригодным для повторного использования. Это случается постоянно, но в некоторых ситуациях в этом нет ничего плохого. Однако мы сейчас ведем речь о проектировании и хотим воспользоваться преимуществами того, что предлагает нам объектно-ориентированный подход. Поэтому наша цель заключается в том, чтобы спроектировать абстрактные, в высокой степени пригодные для повторного применения классы, а для этого мы спроектируем очень абстрактные интерфейсы пользователя.

Для того чтобы вы смогли наглядно увидеть разницу между абстрактным и конкретным интерфейсами, создадим такой объект, как такси. Намного полезнее располагать интерфейсом, например «отвезите меня в аэропорт», чем отдельными интерфейсами, например «поверните направо», «поверните налево», «поехали», «остановитесь» и т. д., поскольку, как показано на рис. 2.4, клиенту нужно лишь одно — добраться до аэропорта.

Когда вы выйдете из отеля, в котором жили, бросите свои чемоданы на заднее сиденье такси и сядете в машину, таксист повернется к вам и спросит: «Куда вас

отвезти?» Вы ответите: «Пожалуйста, отвезите меня в аэропорт» (при этом, естественно, предполагается, что в городе есть только один крупный аэропорт. Например, в Чикаго вы сказали бы: «Пожалуйста, отвезите меня в аэропорт “Мидуэй”» или «Пожалуйста, отвезите меня в аэропорт “О’Хара”»). Вы сами, возможно, даже не будете знать, как добраться до аэропорта, но даже если и будете, то вам не придется рассказывать таксисту о том, когда и в какую сторону нужно повернуть, как показано на рис. 2.5. Каким в действительности путем таксист поедет, для вас как пассажира не будет иметь значения (однако плата за проезд в какой-то момент может стать предметом разногласий, если таксист решит сжульничать и повезти вас в аэропорт длинным путем).

Рис. 2.4. Абстрактный интерфейс

В чем же проявляется связь между абстрактностью и повторным использованием? Задайте себе вопрос насчет того, какой из этих двух сценариев более пригоден для повторного использования — абстрактный или не такой абстрактный? Проще говоря, какая фраза более подходит для того, чтобы использовать ее снова: «Отвезите меня в аэропорт» или «Поверните направо, затем направо, затем налево, затем направо, затем налево»? Очевидно, что первая фраза является более подходящей. Вы можете сказать ее в любом городе всякий раз, когда садитесь в такси и хотите добраться до аэропорта. Вторая фраза подойдет только в отдельных случаях. Таким образом, абстрактный интерфейс «Отвезите меня в аэропорт» в целом является отличным вариантом грамотного подхода к объектно-ориентированному проектированию, результат которого окажется пригоден для повторного использования, а его реализация будет разной в Чикаго, Нью-Йорке и Кливленде.

Рис. 2.5. Не такой абстрактный интерфейс

Обеспечение самого минимального интерфейса пользователя из возможных

При проектировании класса общее правило заключается в том, чтобы всегда обеспечивать для пользователей как можно меньше информации о внутреннем устройстве этого класса. Чтобы сделать это, придерживайтесь следующих простых правил.

- Предоставляйте пользователям только то, что им обязательно потребуется. По сути это означает, что у класса должно быть как можно меньше интерфейсов. Приступая к проектированию класса, начинайте с минимального интерфейса. Проектирование класса итеративно, поэтому вскоре вы обнаружите, что минимального набора интерфейсов недостаточно. И это будет нормально.

  Лучше в дальнейшем добавить интерфейсы из-за того, что они окажутся действительно нужны пользователям, чем сразу давать этим людям больше интерфейсов, нежели им требуется. Иногда доступ конкретного пользователя к определенным интерфейсам создает проблемы. Например, вам не понадобится интерфейс, предоставляющий информацию о заработной плате всем пользователям, — такие сведения должны быть доступны только тем, кому их положено знать.

  Сейчас рассмотрим наш программный пример. Представьте себе пользователя, несущего системный блок персонального компьютера без монитора или клавиатуры. Очевидно, что от этого персонального компьютера будет мало толку. Здесь для пользователя просто предусматривается минимальный набор интерфейсов, относящийся к персональному компьютеру. Однако этого минималь-

ного набора недостаточно, и сразу же возникает необходимость добавить другие интерфейсы.

- Открытые интерфейсы определяют, что у пользователей имеется доступ. Если вы изначально скроете весь класс от пользователей, сделав интерфейсы закрытыми, то, когда программисты начнут применять этот класс, вам придется сделать определенные методы открытыми, которые, таким образом, станут частью открытого интерфейса.
- Жизненно важно проектировать классы с точки зрения пользователя, а не с позиции информационных систем. Слишком часто бывает так, что проектировщики классов (не говоря уже о программном обеспечении всех прочих видов) создают тот или иной класс таким образом, чтобы он вписывался в конкретную технологическую модель. Даже если проектировщик станет все делать с точки зрения пользователя, то такой пользователь все равно окажется скорее техническим специалистом, а класс будет проектироваться с таким расчетом, чтобы он работал с технологической точки зрения, а не был удобным в применении пользователями.
- Занимаясь проектированием класса, перечитывайте соответствующие требования и проектируйте его, общаясь с людьми, которые станут использовать этот класс, а не только с разработчиками. Класс, скорее всего, будет эволюционировать, и его потребуется обновить при создании прототипа системы.

Определение пользователей

Снова обратимся к примеру с такси. Мы уже решили, что пользователи в данном случае — это те люди, которые фактически будут применять соответствующую систему. При этом сам собой напрашивается вопрос: что это за люди?

Первым порывом будет сказать, что ими окажутся *клиенты*. Однако это правильно только примерно наполовину. Хотя клиенты, конечно же, являются пользователями, таксист должен быть способен успешно оказать клиентам соответствующую услугу. Другими словами, обеспечение интерфейса, который, несомненно, понравился бы клиенту, например «Отвезите меня в аэропорт бесплатно», не устроит таксиста. Таким образом, для того, чтобы создать реалистичный и пригодный к использованию интерфейс, следует считать пользователями *как* клиента, *так и* таксиста.

В качестве программной аналогии представьте себе, что пользователям потребовалось, чтобы программист обеспечил определенную функцию. Однако если программист обнаружит, что выполнить это требование технически невозможно, то он не сможет его удовлетворить независимо от того, насколько сильно захочет помочь.

Коротко говоря, любой объект, который отправляет сообщение другому объекту, представляющему такси, считается пользователем (и да, пользователи тоже являются объектами). На рис. 2.6 показано, как таксист оказывает услугу.

ЗАБЕГАЯ ВПЕРЕД

Таксист, скорее всего, тоже будет объектом.

Рис. 2.6. Оказание услуги

Поведения объектов

Определение пользователей — это лишь часть того, что нужно сделать. После того как пользователи будут определены, вам потребуется определить поведения объектов. Исходя из точки зрения всех пользователей, начинайте определение назначения каждого объекта и того, что он должен делать, чтобы работать должным образом. Следует отметить, что многое из того, что будет выбрано первоначально, не приживется в окончательном варианте открытого интерфейса. Нужно определяться с выбором во время сбора требований с применением различных методик, например на основе вариантов использования UML.

Ограничения, налагаемые средой

В своей книге «Объектно-ориентированное проектирование на Java» *(Object-Oriented Design in Java)* Гилберт и Маккарти отмечают, что среда часто налагает ограничения на возможности объекта. Фактически почти всегда имеют место ограничения, налагаемые средой. Компьютерное аппаратное обеспечение может ограничивать функциональность программного обеспечения. Например, система может быть не подключена к сети или в компании может использоваться принтер специфического типа. В примере с такси оно не сможет продолжить путь по дороге, если в соответствующем месте не окажется моста, даже если это будет более короткий путь к аэропорту.

Определение открытых интерфейсов

После того как будет собрана вся информация о пользователях, поведениях объектов и среде, вам потребуется определить открытые интерфейсы для каждого пользовательского объекта. Поэтому подумайте о том, как бы вы использовали такой объект, как такси.

1. Сесть в такси.
2. Сказать таксисту о том, куда вас отвезти.
3. Заплатить таксисту.
4. Дать таксисту на чай.
5. Выйти из такси.

Что вам необходимо для того, чтобы использовать такой объект, как такси?

1. Располагать местом, до которого вам нужно добраться.
2. Подозвать такси.
3. Заплатить таксисту.

Сначала вы задумаетесь о том, как объект используется, а не о том, как он создается. Вы, возможно, обнаружите, что объекту требуется больше интерфейсов, например «Положить чемодан в багажник» или «Вступить в пустой разговор с таксистом». На рис. 2.7 показана диаграмма класса, отражающая возможные методы для класса `Cabbie`.

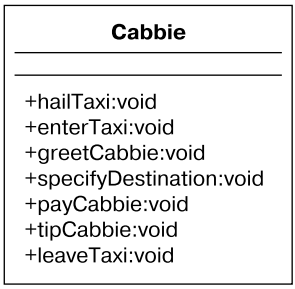

Рис. 2.7. Методы в классе Cabbie

Как и всегда, «шлифовка» финального интерфейса — итеративный процесс. Для каждого интерфейса вам потребуется определить, вносит ли он свой вклад в эксплуатацию объекта. Если нет, то, возможно, он не нужен. Во многих учебниках по объектно-ориентированному программированию рекомендуется, чтобы каждый интерфейс моделировал только одно поведение. Это возвращает нас к вопросу о том, какого абстрагирования мы хотим добиться при проектировании. Если у нас будет интерфейс с именем `enterTaxi()`, то, безусловно, нам не потребуется, чтобы в нем содержалась логика «заплатить таксисту». Если все так и будет, то наша конструкция окажется несколько нелогичной. Кроме того, фактически пользователи класса никак не смогут узнать, что нужно сделать для того, чтобы «заплатить таксисту».

Определение реализации

Выбрав открытые интерфейсы, вы должны будете определить реализацию. После того как вы спроектируете класс, а все методы, необходимые для эксплуатации класса, окажутся на своих местах, потребуется заставить этот класс работать.

Технически все, что не является открытым интерфейсом, можно считать реализацией. Это означает, что пользователи никогда не увидят каких-либо методов, считающихся частью реализации, в том числе подпись конкретного метода (которая включает имя метода и список параметров), а также фактический код внутри этого метода.

Можно располагать закрытым методом, который применяется внутри класса. Любой закрытый метод считается частью реализации при условии, что пользователи

никогда не увидят его и, таким образом, не будут иметь к нему доступа. Например, в классе может содержаться метод `changePassword()`; однако этот же класс может включать закрытый метод для шифрования пароля. Этот метод был бы скрыт от пользователей и вызывался бы только изнутри метода `changePassword()`.

Реализация полностью скрыта от пользователей. Код внутри открытых методов является частью реализации, поскольку пользователи не могут увидеть его (они должны видеть только вызывающую структуру интерфейса, а не код, что находится в нем).

Это означает, что теоретически все считающееся реализацией может изменяться, не влияя на то, как пользователи взаимодействуют с классом. При этом, естественно, предполагается, что реализация дает ответы, ожидаемые пользователями.

Хотя интерфейс представляет то, как пользователи видят определенный объект, в действительности реализация — это основная составляющая этого объекта. Реализация содержит код, который описывает состояние объекта.

Резюме

В этой главе мы исследовали три области, которые помогут вам начать мыслить объектно-ориентированным образом. Помните, что не существует какого-либо постоянного списка вопросов, касающихся объектно-ориентированного мышления. Работа в объектно-ориентированном стиле больше представляет собой искусство, нежели науку. Попробуйте сами придумать, как можно было бы описать объектно-ориентированное мышление.

В главе 3 мы поговорим о жизненном цикле объектов: как они «рождаются», «живут» и «умирают». Пока объект «жив», он может переходить из одного в другое состояние, которых много. Например, объект `DataBaseReader` будет пребывать в одном состоянии, если база данных окажется открыта, и в другом состоянии — если она будет закрыта. Способ, каким все это будет представлено, зависит от того, как окажется спроектирован соответствующий класс.

Ссылки

- *Майерс Скотт.* Эффективное использование C++ (Effective C++). 3-е изд. — Бостон: Addison-Wesley Professional, 2005.
- *Фаулер Мартин.* UML. Основы (UML Distilled). 3-е изд. — Бостон: Addison-Wesley Professional, 2003.
- *Гилберт Стивен и Маккарти Билл.* Объектно-ориентированное проектирование на Java (Object-Oriented Design in Java). — Беркли: The Waite Group Press (Pearson Education), 1998.

Глава 3
Продвинутые объектно-ориентированные концепции

В главах 1 и 2 мы рассмотрели основы объектно-ориентированных концепций. Прежде чем мы приступим к исследованию более деликатных задач проектирования, касающихся создания объектно-ориентированных систем, нам необходимо рассмотреть такие более продвинутые объектно-ориентированные концепции, как конструкторы, перегрузка операторов и множественное наследование. Мы также поговорим о методиках обработки ошибок и важности знания того, как область видимости применима в сфере объектно-ориентированного проектирования.

Некоторые из этих концепций могут не быть жизненно важными для понимания объектно-ориентированного проектирования на более высоком уровне, однако они необходимы любому, кто занят в проектировании и реализации той или иной объектно-ориентированной системы.

Конструкторы

Конструкторы, возможно, не станут новинкой для тех, кто занимается структурным программированием. В некоторых объектно-ориентированных языках вроде Java и C# конструкторы представляют собой методы с тем же именем, что и соответствующий класс. В Visual Basic .NET используется обозначение `New`, а в Objective-C — ключевое слово `init`. Как обычно, мы сосредоточимся на концепциях конструкторов и не станем рассматривать специфический синтаксис всех языков. Взглянем на Java-код, реализующий конструктор.

Например, конструктор для класса `Cabbie`, рассмотренного нами в главе 2, выглядел бы так:

```
public Cabbie(){
  /* код для конструирования объекта */
}
```

Компилятор увидит, что имя метода идентично имени класса, и будет считать этот метод конструктором.

ПРЕДОСТЕРЕЖЕНИЕ

Следует отметить, что в этом Java-коде (как и в написанном на C# и C++) у конструктора нет возвращаемого значения. Если вы предусмотрите возвращаемое значение, то компилятор не будет считать метод конструктором.

Например, если вы включите приведенный далее код в класс, то компилятор не будет считать метод конструктором, поскольку у того есть возвращаемое значение, которым в данном случае является целочисленная величина:

```
public int Cabbie(){
  /* код для конструирования объекта */
}
```

Это синтаксическое требование может создавать проблемы, поскольку этот код пройдет компиляцию, но не будет вести себя так, как это ожидается.

Когда осуществляется вызов конструктора

При создании нового объекта в первую очередь выполняется вызов конструктора. Взгляните на приведенный далее код:

```
Cabbie myCabbie = new Cabbie();
```

Ключевое слово `new` обеспечит создание класса `Cabbie` и таким образом приведет к выделению требуемой памяти. Затем произойдет вызов конструктора как такового с передачей аргументов в списке параметров. Конструктор обеспечивает для разработчиков возможность следить за соответствующей инициализацией.

Таким образом, код `new Cabbie()` создаст экземпляр объекта `Cabbie` и вызовет метод `Cabbie`, который является конструктором.

Что находится внутри конструктора

Пожалуй, наиболее важной функцией конструктора является инициализация выделенной памяти при обнаружении ключевого слова `new`. Коротко говоря, код, заключенный внутри конструктора, должен задать для нового созданного объекта его начальное, стабильное, надежное состояние.

Например, если у вас есть объект `Counter` с атрибутом `count`, то вам потребуется задать для `count` значение `0` в конструкторе:

```
count = 0;
```

ИНИЦИАЛИЗАЦИЯ АТРИБУТОВ

Функция, называемая служебной (или инициализацией), часто используется в целях инициализации при структурном программировании. Инициализация атрибутов представляет собой общую операцию, выполняемую в конструкторе.

Конструктор по умолчанию

Если вы напишете класс и не добавите в него конструктор, то этот класс все равно пройдет компиляцию и вы сможете его использовать. Если в классе не окажется предусмотрено явного конструктора, то будет обеспечен конструктор по умолчанию. Важно понимать, что в наличии всегда будет как минимум один конструктор вне зависимости от того, напишете ли вы его сами. Если вы не предусмотрите конструктора, то система обеспечит за вас конструктор по умолчанию.

Помимо создания объекта как такового, единственное действие, предпринимаемое конструктором по умолчанию, — это вызов конструктора его суперкласса.

Во многих случаях суперкласс будет частью фреймворка языка, как класс `Object` в Java. Например, если окажется, что для класса `Cabbie` не предусмотрено конструктора, то будет добавлен приведенный далее конструктор по умолчанию:

```
public Cabbie(){
  super();
}
```

Если бы вам потребовалось декомпилировать байт-код, выдаваемый компилятором, то вы увидели бы этот код. Его в действительности вставляет компилятор.

В данном случае, если `Cabbie` не наследует явным образом от другого класса, то класс `Object` будет родительским. Пожалуй, в некоторых ситуациях конструктора по умолчанию окажется достаточно; однако в большинстве случаев потребуется инициализация памяти. Независимо от ситуации, правильная методика программирования — всегда включать в класс минимум один конструктор. Если в классе содержатся атрибуты, то желательна их инициализация. Кроме того, инициализация переменных всегда будет правильной методикой при написании кода, будь он объектно-ориентированным или нет.

ОБЕСПЕЧЕНИЕ КОНСТРУКТОРА

Общее правило заключается в том, что вы должны всегда обеспечивать конструктор, даже если не планируете что-либо делать внутри него. Вы можете предусмотреть конструктор, в котором ничего нет, а затем добавить в него что-то. Хотя технически ничего плохого в использовании конструктора по умолчанию, обеспечиваемого компилятором, нет, в целях документирования и сопровождения никогда не будет лишним знать, как именно выглядит ваш код.

Неудивительно, что сопровождение становится здесь проблемой. Если вы зависите от конструктора по умолчанию, а при последующем сопровождении будет добавлен еще один конструктор, то конструктор по умолчанию больше не будет обеспечиваться. Коротко говоря, конструктор по умолчанию добавляется, только если вы сами не включите никаких конструкторов. Как только вы предусмотрите хотя бы один конструктор, конструктор по умолчанию больше не будет обеспечиваться.

Использование множественных конструкторов

Во многих случаях объект можно будет сконструировать несколькими методами. Чтобы приспособиться к таким ситуациям, вам потребуется предусмотреть более одного конструктора. К примеру, взглянем на представленный здесь класс `Count`:

```
public class Count {

  int count;

  public Count(){
    count = 0;
  }
}
```

С одной стороны, мы хотим инициализировать атрибут `count` для отсчета до нуля — мы можем легко сделать это, использовав конструктор для инициализации `count` значением `0`, как показано далее:

```
public Count(){
  count = 0;
}
```

С другой стороны, нам может потребоваться передать параметр инициализации, который позволит задавать для `count` различные числовые значения:

```
public Count (int number){
  count = number;
}
```

Это называется *перегрузкой метода* (перегрузка имеет отношение ко всем методам, а не только к конструкторам). В большинстве объектно-ориентированных языков предусматривается функциональность для перегрузки методов.

Перегрузка методов

Перегрузка позволяет программистам снова и снова использовать один и тот же метод, если его подпись каждый раз отличается. Подпись состоит из имени метода и списка параметров (рис. 3.1).

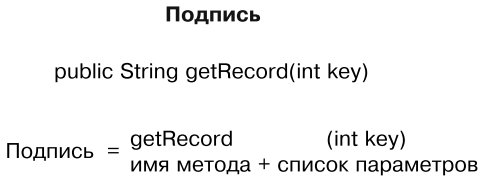

Рис. 3.1. Компоненты подписи

Таким образом, *все* приведенные далее методы имеют разные подписи:

```
public void getCab();

// другой список параметров
public void getCab (String cabbieName);

// другой список параметров
public void getCab (int numberOfPassengers);
```

ПОДПИСИ

Подпись, в зависимости от языка программирования, может включать или не включать возвращаемый тип. В Java и C# возвращаемый тип не является частью подписи. Например, приведенные далее два метода будут конфликтовать, несмотря на то что возвращаемые типы различаются:

```
public void getCab (String cabbieName);
public int getCab (String cabbieName);
```

Наилучший способ понять подписи заключается в том, чтобы написать код и прогнать его через компилятор.

Используя различающиеся подписи, вы можете по-разному конструировать объекты в зависимости от применяемого конструктора. Такая функциональность очень полезна в ситуациях, в которых вы не всегда заранее знаете, сколько данных у вас будет в наличии. Например, при наполнении корзины в интернет-магазине может оказаться так, что клиенты уже вошли под своими учетными записями (и у вас будет вся их информация). С другой стороны, совершенно новый клиент может класть товары в корзину вообще без доступной информации об учетной записи. В каждом из этих случаев конструктор будет по-разному осуществлять инициализацию.

Использование UML для моделирования классов

Вернемся к примеру с `DataBaseReader`, который мы использовали в главе 2. Представим, что у нас есть два способа сконструировать `DataBaseReader`.

- Передать имя базы данных и позицию указателя в начале базы данных.
- Передать имя базы данных и позицию в базе данных, где, как нам необходимо, должен установиться указатель.

На рис. 3.2 приведена диаграмма класса `DataBaseReader`. Обратите внимание на то, что эта диаграмма включает только два конструктора для класса. Хотя на ней показаны два конструктора, без списка параметров нельзя понять, какой конструктор каким именно является. Чтобы провести различие между этими конструкторами, вы можете взглянуть на соответствующий код в классе `DataBaseReader`, приведенный далее.

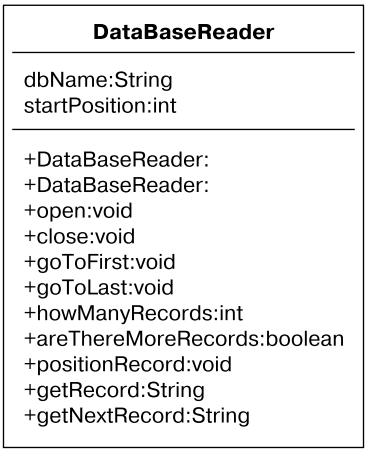

Рис. 3.2. Диаграмма класса DataBaseReader

ОТСУТСТВИЕ ВОЗВРАЩАЕМОГО ТИПА

Обратите внимание, что на диаграмме класса на рис. 3.2 у конструкторов нет возвращаемых типов. У всех прочих методов, кроме конструкторов, должны быть возвращаемые типы.

Вот фрагмент кода класса, который показывает его конструкторы, а также атрибуты, инициализируемые конструкторами (рис. 3.3).

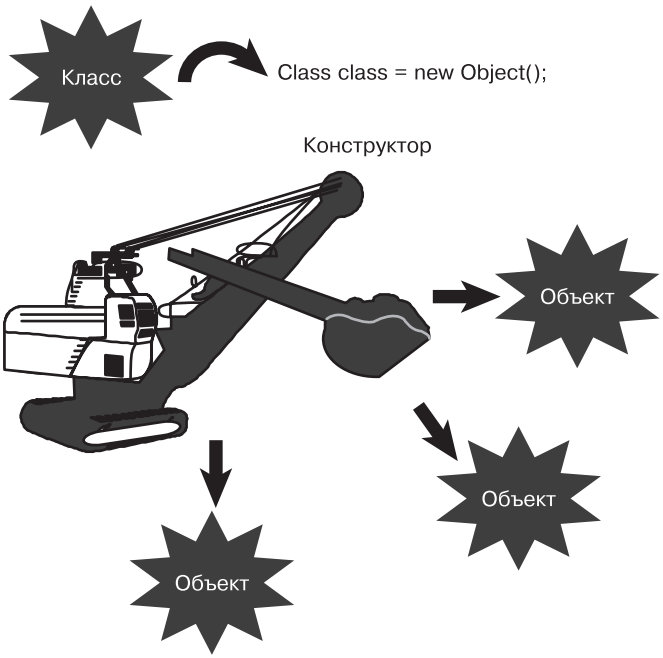

Рис. 3.3. Создание нового объекта

```
public class DataBaseReader {

  String dbName;
  int startPosition;

  // инициализировать только name
  public DataBaseReader (String name){
    dbName = name;
  startPosition = 0;
  };

  // инициализировать name и pos
  public DataBaseReader (String name, int pos){

    dbName = name;
    startPosition = pos;
  };

  .. // остальная часть класса
}
```

Обратите внимание, что инициализация `startPosition` осуществляется в обоих случаях. Если не передать конструктору данные в виде списка параметров, то он будет инициализирован таким значением по умолчанию, как `0`.

Как сконструирован суперкласс

При использовании наследования вы должны знать, как сконструирован соответствующий родительский класс. Помните, что, когда оно задействуется, от родительского класса наследуется все. Таким образом, вам потребуется очень хорошо знать все данные и поведения родительского класса. Наследование атрибутов довольно очевидно. Однако то, как наследуются конструкторы, не так очевидно. После обнаружения ключевого слова `new` и выделения памяти для объекта предпринимаются следующие шаги (рис. 3.4).

1. Внутри конструктора происходит вызов конструктора суперкласса соответствующего класса. Если явного вызова конструктора суперкласса нет, то автоматически вызывается конструктор по умолчанию. При этом вы сможете увидеть соответствующий код, взглянув на байт-коды.
2. Инициализируется каждый атрибут класса объекта. Эти атрибуты являются частью определения класса (переменные экземпляра), а не атрибутами внутри конструктора или любого другого метода (локальные переменные). В коде `DataBaseReader`, показанном ранее, целочисленная переменная `startPosition` является переменной экземпляра класса.
3. Выполняется остальная часть кода внутри конструктора.

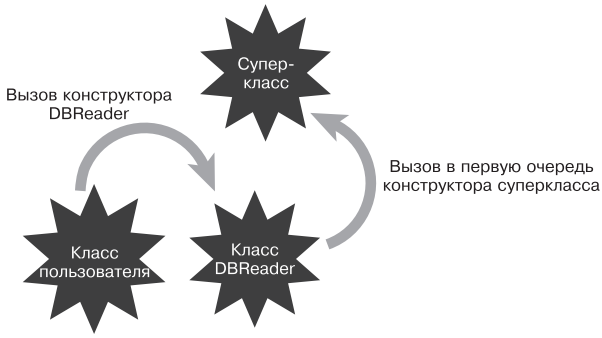

Рис. 3.4. Конструирование объекта

Проектирование конструкторов

Как вы уже видели ранее, при проектировании класса желательна инициализация всех атрибутов. В отдельных языках программирования компилятор обеспечивает некоторую инициализацию. Но, как и всегда, не следует рассчитывать на компилятор

в плане инициализации атрибутов! При использовании Java вы не сможете задействовать тот или иной атрибут до тех пор, пока он не будет инициализирован. Если атрибут впервые задается в коде, то позаботьтесь о том, чтобы инициализировать его с каким-нибудь допустимым условием — например, определить для целочисленной переменной значение 0.

Конструкторы используются для обеспечения того, что приложения будут пребывать в стабильном состоянии (мне нравится называть его «надежным» состоянием). Например, инициализируя атрибут значением 0, можно получить нестабильное приложение, если этот атрибут предназначается для использования в качестве делителя в операции деления. Вы должны учитывать, что деление на нуль — недопустимая операция. Инициализация значением 0 не всегда оказывается наилучшим вариантом.

При проектировании правильная методика заключается в том, чтобы определить стабильное состояние для всех атрибутов, а затем инициализировать их с этим стабильным состоянием в конструкторе.

Обработка ошибок

Крайне редко бывает так, что тот или иной класс оказывается идеально написанным с первого раза. В большинстве, если не во всех ситуациях, *будут* ошибки. Любой разработчик, не имеющий плана действий на случай возникновения проблем, рискует.

Если ваш код способен выявлять и перехватывать ошибочные условия, то вы можете обрабатывать ошибки несколькими путями: в книге «Учебник по Java для начинающих» *(Java Primer Plus)* Пол Тима (Paul Tyma), Габриэл Торок (Gabriel Torok) и Трой Даунинг (Troy Downing) утверждают, что существует три основных подхода к проблемам, выявляемым в программах: устранить проблемы, игнорировать проблемы, отбросив их, или выйти из среды выполнения неким корректным образом. В книге «Объектно-ориентированное проектирование на Java» *(Object-Oriented Design in Java)* Гилберт и Маккарти более подробно останавливаются на этой теме, добавляя такой вариант, как возможность выбрасывать исключения.

- Игнорирование проблем — плохая идея!
- Проверка на предмет проблем и прерывание выполнения программы при их обнаружении.
- Проверка на предмет потенциальных проблем, перехват ошибок и попытка решить обнаруженные проблемы.
- Выбрасывание исключений (оно зачастую оказывается предпочтительным способом урегулирования соответствующих ситуаций).

Эти стратегии рассматриваются в приведенных далее разделах.

Игнорирование проблем

Если просто игнорировать потенциальные проблемы, то это будет залогом провала. Кроме того, если вы собираетесь игнорировать проблемы, то зачем вообще тратить силы на их выявление? Ясно, что вам не следует игнорировать любые из известных проблем. Основная задача для всех приложений заключается в том, что они никогда

не должны завершаться аварийно. Если вы не станете обрабатывать возникшие у вас ошибки, то работа приложения в конечном счете завершится некорректно либо продолжится в режиме, который можно будет считать нестабильным. Во втором случае вы, возможно, даже не будете знать, что получаете неверные результаты, а это может оказаться намного хуже аварийного завершения программы.

Проверка на предмет проблем и прерывание выполнения приложения

Если вы выберете проверку на предмет проблем и прерывание выполнения приложения при их выявлении, то приложение сможет вывести сообщение о наличии неполадок. При этом работа приложения завершится корректно, а пользователю останется смотреть в монитор компьютера, качать головой и задаваться вопросом о том, что произошло. Хотя это намного лучший вариант, чем игнорирование проблем, он никоим образом не является оптимальным. Однако он позволяет системе навести порядок и привести себя в более стабильное состояние, например закрыть файлы и форсировать перезагрузку системы.

Проверка на предмет проблем и попытка устранить неполадки

Проверка на предмет потенциальных проблем, перехват ошибок и попытка устранить неполадки гораздо лучше, чем просто проверять на предмет проблем и прерывать выполнение приложения в соответствующих ситуациях. В данном случае проблемы выявляются кодом, а приложение пытается «починить» себя. Это хорошо работает в определенных ситуациях.

Взгляните, к примеру, на следующий код:

```
if (a == 0)
  a=1;
c = b/a;
```

Ясно, что, если не включить в код условный оператор, а нуль будет стоять после оператора деления, вы получите системное исключение, поскольку делить на нуль нельзя. Если перехватить исключение и задать для переменной значение `1`, то по крайней мере не произойдет фатального сбоя системы. Однако присвоение значения `1` не обязательно поможет, поскольку результат может оказаться неверным. Оптимальное решение состоит в том, чтобы предложить пользователю заново ввести правильное входное значение.

СМЕШЕНИЕ МЕТОДИК ОБРАБОТКИ ОШИБОК

Хотя такая обработка ошибок не обязательно будет объектно-ориентированной по своей природе, я считаю, что у нее есть законное место в ООП. Выбрасывание исключений (о чем пойдет речь в следующем разделе) может оказаться весьма затратным в плане «накладных расходов». Таким образом, даже если исключения могут быть правильным выбором при проектировании, вам все равно необходимо принимать во внимание другие методики обработки ошибок (хотя бы проверенные структурные методики) в зависимости от ваших требований в области проектирования и производительности.

Хотя упоминавшиеся ранее методики выявления ошибок предпочтительнее бездействия, у них тем не менее есть несколько недостатков. Не всегда легко определить, где именно впервые возникла проблема. Кроме того, на выявление проблемы может потребоваться некоторое время. Так или иначе, более подробное объяснение обработки ошибок лежит вне рамок этой книги. Однако при проектировании важно предусматривать в классах обработку ошибок с самого начала, а операционная система зачастую сама может предупреждать вас о проблемах, которые выявляет.

Выбрасывание исключений

В большинстве объектно-ориентированных языков программирования предусматривается такая функция, как *исключения*. В самом общем смысле под исключениями понимаются неожиданные события, которые имеют место в системе. Исключения дают возможность выявлять проблемы, а затем решать их. В Java, C#, C++, Objective-C и Visual Basic исключения обрабатываются при использовании ключевых слов `catch` и `throw`. Это может показаться игрой в бейсбол, однако ключевая концепция в данном случае заключается в том, что пишется определенный блок кода для обработки определенного исключения. Такая методика позволяет выяснить, где проблема берет свое начало, и раскрутить код до соответствующей точки.

Вот структура для Java-блока `try/catch`:

```
try {

  // возможный сбойный код

} catch(Exception e) {

  // код для обработки исключения
}
```

При выбрасывании исключения в блоке `try` оно будет обработано блоком `catch`. Если выбрасывание исключения произойдет, когда блок будет выполняться, то случится следующее.

1. Выполнение блока `try` завершится.
2. Предложения `catch` будут проверены с целью выяснить, надлежащий ли блок `catch` был включен для обработки проблемного исключения (на каждый блок `try` может приходиться более одного предложения `catch`).
3. Если ни одно из предложений `catch` не обработает проблемное исключение, то оно будет передано следующему блоку `try` более высокого уровня (если исключение не будет перехвачено в коде, то система в конечном счете сама перехватит его, а результат будет непредсказуемым, то есть случится аварийное завершение приложения).
4. Если будет выявлено соответствующее предложение `catch` (обнаружено первое из соответствующих), то будут выполнены операторы в предложении `catch`.
5. Выполнение возобновится с оператора, следующего за блоком `try`.

Достаточно сказать, что исключения — серьезное преимущество объектно-ориентированных языков программирования. Вот пример того, как исключение перехватывается при использовании Java:

```
try {

  // возможный сбойный код
  count = 0;
  count = 5/count;

} catch(ArithmeticException e) {

  // код для обработки исключения
  System.out.println(e.getMessage());
  count = 1;

}
System.out.println("Исключение обработано.");
```

СТЕПЕНЬ ДЕТАЛИЗАЦИИ ПРИ ПЕРЕХВАТЕ ИСКЛЮЧЕНИЙ

Вы можете перехватывать исключения с различной степенью детализации. Допускается перехватывать все исключения или проводить проверку на предмет определенных исключений, например арифметических. Если ваш код не будет перехватывать исключения, то это станет делать среда выполнения Java, от чего она не будет в восторге!

В этом примере деление на нуль (поскольку значением `count` является `0`) в блоке `try` приведет к арифметическому исключению. Если исключение окажется сгенерировано (выброшено) вне блока `try`, то программа, скорее всего, завершится (аварийно). Однако, поскольку исключение будет выброшено в блоке `try`, блок `catch` подвергнется проверке с целью выяснить, все ли запланировано на случай возникновения соответствующего исключения (в рассматриваемой нами ситуации оно является арифметическим). Поскольку блок `catch` включает проверку на предмет арифметического исключения, код, содержащийся в этом блоке, выполнится и, таким образом, `count` будет присвоено значение `0`. После того как выполнится блок `catch`, будет осуществлен выход из блока `try/catch`, а в консоли Java появится сообщение `Исключение обработано`. Логическая последовательность этого процесса проиллюстрирована на рис. 3.5.

Рис. 3.5. Перехват исключения

Если вы не поместите `ArithmeticException` в блок `catch`, то программа, вероятно, завершится аварийно. Вы сможете перехватывать все исключения благодаря коду, который приведен далее:

```
try {

  // возможный сбойный код

} catch(Exception e) {

  // код для обработки исключения
}
```

Параметр `Exception` в блоке `catch` используется для перехвата всех исключений, которые могут быть сгенерированы в блоке `try`.

ОШИБКОУСТОЙЧИВЫЙ КОД

Хорошая идея — комбинировать описанные здесь методики для того, чтобы сделать программу как можно более ошибкоустойчивой при ее применении пользователями.

Важность области видимости

Экземпляры множественных объектов могут создаваться на основе одного класса. Каждый из этих объектов будет обладать уникальным идентификатором и состоянием. Это важно. Каждому объекту, конструируемому отдельно, выделяется его собственная отдельная память. Однако если некоторые атрибуты и методы объявлены соответствующим образом, они могут совместно использоваться всеми объектами, экземпляры которых созданы на основе одного и того же класса, и, таким образом, при этом будет совместно использоваться память, выделенная для этих атрибутов и методов класса.

СОВМЕСТНО ИСПОЛЬЗУЕМЫЙ МЕТОД

Конструктор — это хороший пример метода, совместно используемого всеми экземплярами класса.

Методы представляют поведения объекта, а его состояние представляют атрибуты. Существуют атрибуты трех типов:

- локальные;
- атрибуты объектов;
- атрибуты классов.

Локальные атрибуты

Локальные атрибуты принадлежат определенному методу. Взгляните на приведенный далее код:

```
public class Number {

  public method1() {
```

```
    int count;

  }

  public method2() {

  }

}
```

Метод `method1` содержит локальную переменную с именем `count`. Эта целочисленная переменная доступна только внутри метода `method1`. Метод `method2` даже понятия не имеет, что целочисленная переменная `count` существует.

На этом этапе мы познакомимся с очень важной концепцией — областью видимости. Атрибуты (и методы) существуют в определенной области видимости. В данном случае целочисленная переменная `count` существует в области видимости `method1`. При использовании Java, C#, C++ и Objective-C область видимости обозначается фигурными скобками (`{}`). В классе `Number` имеется несколько возможных областей видимости — начните сопоставлять их с соответствующими фигурными скобками.

У класса как такового есть своя область видимости. Каждый экземпляр класса (то есть каждый объект) обладает собственной областью видимости. У методов `method1` и `method2` тоже имеются собственные области видимости. Поскольку целочисленная переменная `count` заключена в фигурные скобки `method1`, при вызове этого метода создается ее копия. Когда выполнение `method1` завершается, копия `count` удаляется.

Чтобы стало еще веселее, взгляните на этот код:

```
public class Number {

  public method1() {
    int count;
  }

  public method2() {
    int count;
  }

}
```

В этом примере есть две копии целочисленной переменной `count` в соответствующем классе. Помните, что `method1` и `method2` обладают собственными областями видимости. Таким образом, компилятор будет знать, к какой копии `count` нужно обращаться, просто выяснив, в каком методе она располагается. Вы можете представлять себе это следующим образом:

```
method1.count;

method2.count;
```

Что касается компилятора, то различить два атрибута ему не составит труда, даже если у них одинаковые имена. Это почти аналогично ситуации, когда у двух человек одинаковая фамилия, но, зная их имена, вы понимаете, что это две отдельные личности.

Атрибуты объектов

Во многих ситуациях при проектировании атрибут должен совместно использоваться несколькими методами в одном и том же объекте. На рис. 3.6, к примеру, показано, что три объекта сконструированы на основе одного класса. Взгляните на приведенный далее код:

```
public class Number {

  int count;    // доступ к переменной имеется у обоих: method1 и method2

  public method1() {
    count = 1;
  }

  public method2() {
    count = 2;
  }

}
```

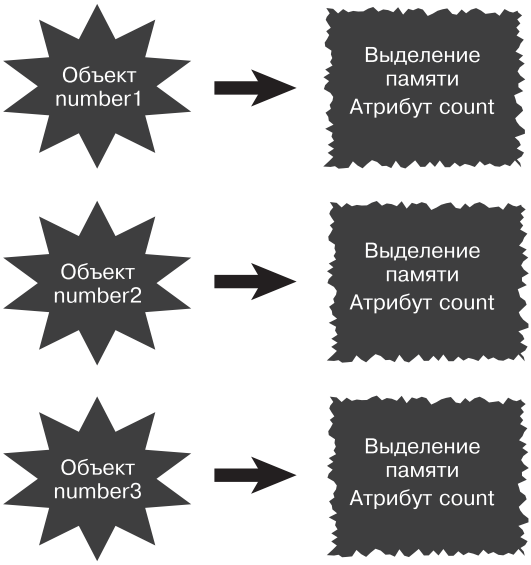

Рис. 3.6. Атрибуты объектов

Обратите здесь внимание на то, что атрибут класса `count` объявлен вне области видимости как `method1`, так и `method2`. Однако он находится в области видимости

класса. Таким образом, атрибут count доступен для обоих method1 и method2 (по сути у всех методов в классе имеется доступ к этому атрибуту). Следует отметить, что в коде, касающемся обоих методов, count присваивается определенное значение. На весь объект приходится только одна копия count, поэтому оба присваивания влияют на одну и ту же копию в памяти. Однако эта копия count не используется совместно разными объектами.

В качестве наглядного примера создадим три копии класса Number:

```
Number number1 = new Number();
Number number2 = new Number();
Number number3 = new Number();
```

Каждый из этих объектов — number1, number2 и number3 — конструируется отдельно, и ему выделяются его собственные ресурсы. Имеется три отдельных экземпляра целочисленной переменной count. Изменение значения атрибута count объекта number1 никоим образом не повлияет на копию count в объекте number2 или number3. В данном случае целочисленная переменная count является *атрибутом объекта*.

Вы можете поиграть в кое-какие интересные игры с областью видимости. Взгляните на приведенный далее код:

```
public class Number {

  int count;
  public method1() {
    int count;
  }

  public method2() {
    int count;
  }

}
```

В данной ситуации в трех полностью отдельных областях памяти для каждого объекта имеется имя count. Объекту принадлежит одна копия, а у method1() и method2() тоже есть по соответствующей копии.

Для доступа к объектной переменной изнутри одного из методов, например method1(), вы можете использовать указатель с именем this из основанных на C языков программирования:

```
public method1() {
  int count;

  this.count = 1;
}
```

Обратите внимание, что часть кода выглядит немного странно:

```
this.count = 1;
```

Выбор слова this в качестве ключевого, возможно, является неудачным. Однако мы должны смириться с ним. Ключевое слово this нацеливает компилятор на доступ к объектной переменной count, а не к локальным переменным в телах методов.

ПРИМЕЧАНИЕ

Ключевое слово this — это ссылка на текущий объект.

Атрибуты классов

Как уже отмечалось ранее, атрибуты могут совместно использоваться двумя и более объектами. При написании кода на Java, C#, C++ или Objective-C для этого потребуется сделать атрибут *статическим*:

```
public class Number {

  static int count;

  public method1() {
  }
}
```

Из-за объявления атрибута `count` статическим ему выделяется один блок памяти для всех объектов, экземпляры которых будут созданы на основе соответствующего класса. Таким образом, все объекты класса будут применять одну и ту же область памяти для `count`. По сути, у каждого класса будет единственная копия, совместно используемая всеми объектами этого класса (рис. 3.7). Это почти настолько близко к глобальным данным, насколько представляется возможным при объектно-ориентированном проектировании.

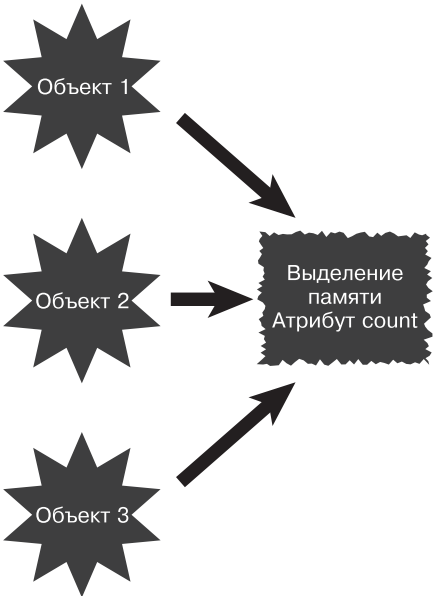

Рис. 3.7. Атрибуты классов

Есть много допустимых вариантов использования атрибутов классов; тем не менее вам следует знать о возможных проблемах синхронизации. Создадим два экземпляра объекта `Count`:

```
Count Count1 = new Count();
Count Count2 = new Count();
```

Теперь представим, что объект `Count1` оживленно занимается своими делами, используя при этом `count` как средство для подсчета пикселов на мониторе компьютера. Это не будет проблемой до тех пор, пока объект `Count2` не решит использовать атрибут `count` для подсчета овец. В тот момент, когда `Count2` запишет данные о первой овце, информация, сохраненная `Count1`, будет потеряна.

Перегрузка операторов

Некоторые объектно-ориентированные языки программирования позволяют выполнять перегрузку операторов. Пример одного из таких языков программирования — C++. Перегрузка операторов дает возможность изменять их смысл. Например, когда большинство людей видит знак плюса, они предполагают, что он означает операцию сложения. Если вы увидите уравнение:

```
X = 5 + 6;
```

то решите, что `X` будет содержать значение `11`. И в этом случае вы окажетесь правы.

Однако иногда знак плюса может означать кое-что другое. Как, например, в следующем коде:

```
String firstName = "Joe", lastName = "Smith";
String Name = firstName + " " + lastName;
```

Вы ожидали бы, что `Name` будет содержать `Joe Smith`. Знак плюса здесь был перегружен для выполнения конкатенации строк.

КОНКАТЕНАЦИЯ СТРОК

Конкатенация строк происходит, когда две отдельные строки объединяют, чтобы создать новую, единую строку.

В контексте строк знак плюса означает не операцию сложения целых чисел или чисел с плавающей точкой, а конкатенацию строк.

А как насчет сложения матриц? Мы могли бы написать такой код:

```
Matrix a, b, c;

c = a + b;
```

Таким образом, знак плюса обеспечивает здесь сложение матриц, а не сложение целых чисел или чисел с плавающей точкой.

Перегрузка — это мощный механизм. Однако он может откровенно сбивать с толку тех, кто читает и сопровождает код. Фактически разработчики могут сами себя запутать. Доводя это до крайности, отмечу, что можно было бы изменить операцию сложения на операцию вычитания. А почему бы и нет? Перегрузка операторов позволяет нам изменять их смысл. Таким образом, если внести изменение, благодаря которому знак плюса будет обеспечивать вычитание, то результатом выполнения приведенного далее кода станет значение `X`, равное `-1`:

```
x = 5 + 6;
```

Более поздние объектно-ориентированные языки программирования, например Java, .NET и Objective-C, не позволяют выполнять перегрузку операторов. Несмотря на это, они сами перегружают знак плюса для конкатенации строк, но дело этим и ограничивается. Люди, разрабатывавшие Java, должно быть, решили, что перегрузка операторов — овчинка, не стоящая выделки. Если вам потребуется выполнять перегрузку операторов при программировании на C++, то позаботьтесь о соответствующем документировании и комментировании, чтобы люди, которые будут использовать ваш класс, не запутались.

Множественное наследование

Намного подробнее о наследовании мы поговорим в главе 7. А эта глава хорошо подходит для того, чтобы начать рассмотрение множественного наследования, которое представляет собой один из наиболее важных и сложных аспектов проектирования классов.

Как видно из названия, *множественное наследование* позволяет тому или иному классу наследовать более чем от одного класса. Это кажется отличной идеей. Объекты должны моделировать реальный мир, не так ли? При этом существует много реальных примеров множественного наследования. Родители — хороший пример такого наследования. У каждого ребенка есть два родителя — таков порядок. Поэтому ясно, что вы можете проектировать классы с применением множественного наследования. Для этого вы сможете использовать некоторые объектно-ориентированные языки программирования, например C++.

Однако эта ситуация подпадает под категорию тех, что схожи с перегрузкой операторов. Множественное наследование — это очень мощная методика, и фактически некоторые проблемы довольно трудно решить без нее. Множественное наследование даже позволяет изящно решать отдельные проблемы. Но оно может значительно усложнить систему как для программистов, так и для разработчиков компиляторов.

Как и в случае с перегрузкой операторов, люди, проектировавшие Java, .NET и Objective-C, решили, что увеличение сложности из-за предоставления возможности использовать множественное наследование значительно перевешивает его преимущества, поэтому они исключили эту функцию из соответствующих языков программирования. В какой-то мере языковые конструкции интерфейсов Java, .NET и Objective-C компенсируют это; однако важно то, что данные языки не позволяют использовать обычное множественное наследование.

ПОВЕДЕНЧЕСКОЕ НАСЛЕДОВАНИЕ И НАСЛЕДОВАНИЕ РЕАЛИЗАЦИИ

Интерфейсы — механизм для поведенческого наследования, в то время как абстрактные классы используются для наследования реализации. Основной момент заключается в том, что языковые конструкции интерфейсов обеспечивают поведенческие интерфейсы, но не реализацию, тогда как абстрактные классы могут обеспечивать как интерфейсы, так и реализацию. Более подробно эта тема рассматривается в главе 8.

Операции с объектами

Некоторые самые простые операции в программировании усложняются, когда вы имеете дело с комплексными структурами данных и объектами. Например, если вам потребуется скопировать или сравнить примитивные типы данных, то соответствующий процесс будет довольно простым. Однако копирование и сравнение объектов окажется уже не настолько простым. В своей книге «Эффективное использование C++» *(Effective C++)* Скотт Майерс посвятил целый раздел копированию и присваиванию объектов.

КЛАССЫ И ССЫЛКИ

Проблема с комплексными структурами данных и объектами состоит в том, что они могут содержать ссылки. Просто скопировав ссылку, вы не скопируете структуры данных или объект, к которому она ведет. В том же духе при сравнении объектов, просто сопоставив один указатель с другим, вы сравните только ссылки, а не то, на что они указывают.

Проблемы возникают при сравнении и копировании объектов. В частности, вопрос сводится к тому, станете ли вы следовать указателям. Независимо от этого должен быть способ скопировать объект. Опять-таки эта операция не будет такой простой, какой она может показаться. Поскольку объекты могут содержать ссылки, потребуется придерживаться соответствующих деревьев ссылок для того, чтобы выполнить правильное копирование (если вам действительно понадобится выполнить глубокое копирование).

ГЛУБОКОЕ ИЛИ ПОВЕРХНОСТНОЕ КОПИРОВАНИЕ?

Глубокое копирование происходит, когда вы следуете всем ссылкам, а новые копии создаются для всех объектов, на которые имеются ссылки. В глубокое копирование может вовлекаться много уровней. Если у вас есть объекты со ссылками на множество объектов, которые, в свою очередь, могут содержать ссылки на еще большее количество объектов, то сама копия может требовать значительных «накладных расходов». При поверхностном копировании просто будет сделана копия ссылки без следования уровням. Гилберт и Маккарти хорошо пояснили то, что представляют собой поверхностная и глубокая иерархии, в книге «Объектно-ориентированное проектирование на Java» (Object-Oriented Design in Java).

В качестве наглядного примера взгляните на рис. 3.8, где показано, что если сделать простую копию объекта (называемую *побитовой*), то будут скопированы только ссылки и ни один из фактических объектов. Таким образом, оба объекта (оригинал и копия) будут ссылаться (указывать) на одни и те же объекты. Чтобы выполнить полное копирование, при котором будут скопированы все ссылочные объекты, вам потребуется написать код для создания всех подобъектов.

Подобная проблема также проявляется при сравнении объектов. Как и функция копирования, эта операция не будет такой простой, какой она может показаться.

Поскольку объекты содержат ссылки, потребуется придерживаться соответствующих деревьев ссылок для того, чтобы правильно сравнить эти объекты. В большинстве случаев языки программирования по умолчанию обеспечивают механизм для сравнения объектов. Но, как это обычно бывает, не следует полагаться на такой механизм. При проектировании класса вы должны рассмотреть возможность обеспечения в нем функции сравнения, которая будет работать так, как вам необходимо.

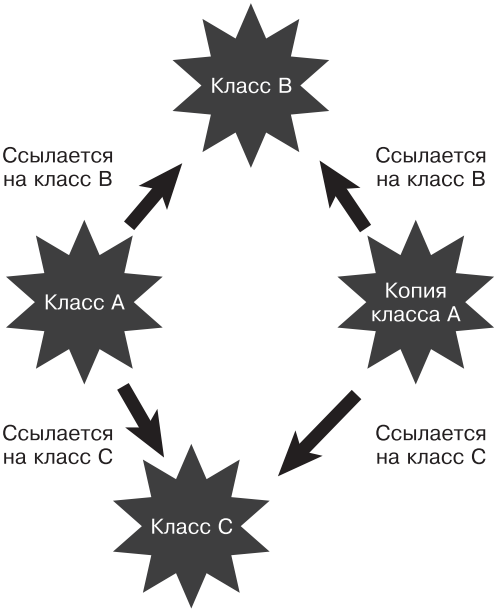

Рис. 3.8. Следование объектным ссылкам

Резюме

В этой главе мы рассмотрели несколько продвинутых объектно-ориентированных концепций, которые, возможно, и не имеют жизненно важного значения для общего понимания остальных объектно-ориентированных концепций, но крайне необходимы для решения объектно-ориентированных задач более высокого уровня, таких, например, как проектирование классов. В главе 4 мы приступим к рассмотрению того, как проектируются и создаются классы.

Ссылки

- *Майерс Скотт.* Эффективное использование C++ (Effective C++). 3-е изд. — Бостон: Addison-Wesley Professional, 2005.
- *Гилберт Стивен и Маккарти Билл.* Объектно-ориентированное проектирование на Java (Object-Oriented Design in Java). — Беркли: The Waite Group Press (Pearson Education), 1998.

- *Тима Пол, Торок Габриэл и Даунинг Трой.* Учебник по Java для начинающих (Java Primer Plus). — Беркли: The Waite Group, 1996.

Примеры кода, использованного в этой главе

Приведенный далее код написан на C# .NET. Эти примеры соответствуют Java-коду, продемонстрированному в текущей главе.

Пример TestNumber: C# .NET

```
using System;

namespace TestNumber
{
  class Program
  {
    public static void Main()
    {

      Number number1 = new Number();
      Number number2 = new Number();
      Number number3 = new Number();
    }
  }

  public class Number
  {
    int count = 0;  // доступ к переменной имеется у обоих: method1 и method2

    public void method1()
    {
      count = 1;
    }

    public void method2()
    {
      count = 2;
    }
  }
}
```

Глава 4
Анатомия класса

В предыдущих главах мы рассмотрели основные объектно-ориентированные концепции и выяснили разницу между интерфейсом и реализацией. Неважно, насколько хорошо вы продумаете, что должно быть частью интерфейса, а что — частью реализации, самое важное всегда в итоге будет сводиться к тому, насколько полезным является определенный класс и как он взаимодействует с другими классами. Никогда не проектируйте класс «в вакууме», или, как можно было бы выразиться, один класс в поле не воин. При создании экземпляров объектов они почти всегда взаимодействуют с другими объектами. Тот или иной объект также может быть частью другого объекта либо частью иерархии наследования.

В этой главе мы рассмотрим простой класс, разобрав его по частям. Кроме того, я буду приводить руководящие указания, которые вам следует принимать во внимание при проектировании классов. Мы продолжим использовать пример с таксистом, описанный в главе 2.

В каждом из следующих разделов рассматривается определенная часть класса. Хотя не все компоненты необходимы в каждом классе, важно понимать то, как классы проектируются и конструируются.

ПРИМЕЧАНИЕ

Рассматриваемый здесь класс призван послужить лишь наглядным примером. Некоторые методы не воплощены в жизнь (то есть их реализация отсутствует) и просто представляют интерфейс — главным образом с целью подчеркнуть то, что интерфейс является «центром» исходной конструкции.

Имя класса

Имя класса важно по нескольким причинам. Вполне понятная из них заключается в том, что имя идентифицирует класс как таковой. Помимо того, чтобы просто идентифицировать класс, имя должно быть описательным. Выбор имени важен, ведь оно обеспечивает информацию о том, что класс делает и как он взаимодействует в рамках более крупных систем.

Имя также важно, если принять во внимание ограничения используемого языка программирования. Например, в Java имя открытого класса должно быть аналогично файловому имени. Если эти имена не совпадут, то приложение не будет работать.

На рис. 4.1 показан класс, который мы будем исследовать. Имя этого класса, понятное и простое, звучит как Cabbie и указывается после ключевого слова `class`:

```
public class Cabbie {
}
```

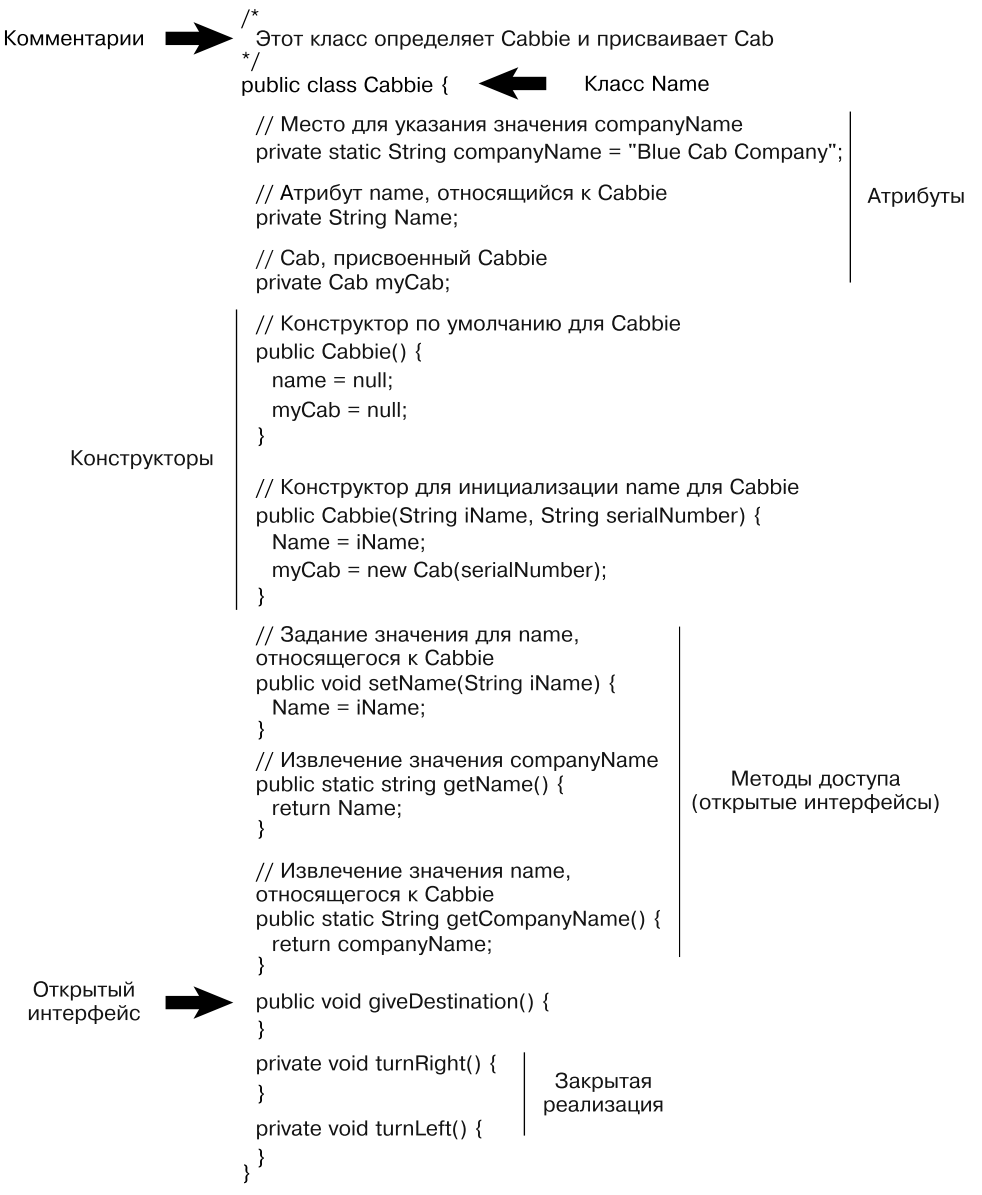

Рис. 4.1. Класс, используемый в качестве примера

ИСПОЛЬЗОВАНИЕ JAVA-СИНТАКСИСА

Помните, что в этой книге мы используем Java-синтаксис. Он будет в чем-то похожим, но все равно несколько иным в случае применения C#, .NET, VB .NET, Objective-C или C++ и абсолютно другим при использовании прочих объектно-ориентированных языков программирования, например Smalltalk.

Имя класса `Cabbie` будет использоваться каждый раз при создании экземпляра этого класса.

Комментарии

Независимо от используемого синтаксиса комментариев они жизненно важны для понимания функции классов. В Java, C# .NET, Objective-C и C++ комментарии бывают двух типов.

ДОПОЛНИТЕЛЬНЫЙ ТИП КОММЕНТАРИЕВ В JAVA И C#

В Java и C# имеется три типа комментариев. В Java третий тип комментариев (/** */) относится к форме документирования, предусматриваемой этим языком программирования. Этот тип комментариев не рассматривается в данной книге. C# предусматривает похожий синтаксис для создания XML-документов.

Первый комментарий оформлен в старомодном стиле языка программирования C и предполагает использование /* (слэш, звездочка) для открытия комментария и */ (звездочка, слэш) для закрытия комментария. Комментарий этого типа может растягиваться более чем на одну строку, и важно не забывать использовать пару открывающих и закрывающих символов в каждом таком комментарии. Если вы не укажете пару закрывающих символов комментария (*/), то какая-то часть вашего кода может оказаться помеченной как комментарий и игнорироваться компилятором. Вот пример комментария этого типа, использованного для класса `Cabbie`:

```
/*

  Этот класс определяет Cabbie и присваивает Cab

*/
```

Второй тип комментария предполагает использование // (слэш, слэш). При этом все, что располагается после двойного слэша до конца строки, считается комментарием. Комментарий такого типа размещается только на одной строке, поэтому вам не нужно указывать закрывающий символ комментария. Вот пример комментария этого типа, использованного для класса `Cabbie`:

```
// Атрибут name, относящийся к Cabbie
```

Атрибуты

Атрибуты представляют состояние определенного объекта, поскольку в них содержится информация об этом объекте. В нашем случае класс `Cabbie` включает

атрибуты `companyName` и `name`, содержащие соответствующие значения, а также `Cab`, присвоенный `Cabbie`. Например, первый атрибут с именем `companyName` хранит следующее значение:

```
private static String companyName = "Blue Cab Company";
```

Обратите внимание на два ключевых слова: `private` и `static`. Ключевое слово `private` означает, что доступ к методу или переменной возможен только внутри объявляемого объекта.

СКРЫТИЕ КАК МОЖНО БОЛЬШЕГО КОЛИЧЕСТВА ДАННЫХ

Все атрибуты в этом примере являются закрытыми. Это соответствует принципу проектирования, согласно которому интерфейс следует проектировать таким образом, чтобы он получился самым минимальным из возможных. Единственный способ доступа к этим атрибутам — воспользоваться методом, обеспечиваемым интерфейсами (на эту тему мы поговорим позднее в текущей главе).

Ключевое слово `static` означает, что на все объекты, экземпляры которых будут созданы на основе соответствующего класса, будет приходиться только одна копия этого атрибута. По сути это атрибут класса (см. главу 3, где атрибуты классов рассматриваются более подробно). Таким образом, даже если на основе класса `Cabbie` будут созданы экземпляры 500 объектов, в памяти окажется только одна копия атрибута `companyName` (рис. 4.2).

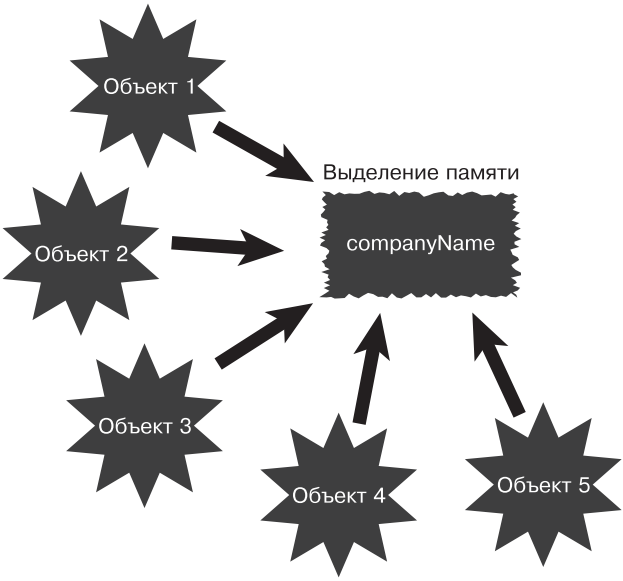

Рис. 4.2. Выделение памяти для объектов

Второй атрибут с именем `name` представляет собой строку, содержащую соответствующее значение `Cabbie`:

```
private String name;
```

Этот атрибут тоже является закрытым, поэтому у других объектов не будет прямого доступа к нему. Им потребуется использовать методы интерфейсов.

Атрибут `myCab` — это ссылка на другой объект. Класс с именем `Cab` содержит информацию о такси вроде его регистрационного номера и журнала технического обслуживания:

```
private Cab myCab;
```

ПЕРЕДАЧА ССЫЛКИ

Объект Cab наверняка был создан другим объектом. Таким образом, объектная ссылка была бы передана объекту Cabbie. Однако в рассматриваемом нами примере Cab был создан внутри объекта Cabbie. В результате нас не очень интересует внутреннее устройство объекта Cab.

Следует отметить, что на этом этапе была создана только ссылка на объект `Cab`; это определение не привело к выделению памяти.

Конструкторы

Класс `Cabbie` содержит два конструктора. Мы знаем, что это именно конструкторы, поскольку у них то же имя, что и у соответствующего класса, — `Cabbie`. Первый конструктор — это конструктор по умолчанию:

```
public Cabbie() {

  name = null;
  myCab = null;

}
```

Технически это не конструктор по умолчанию, обеспечиваемый системой. Напомню, что компилятор обеспечит конструктор по умолчанию, если вы не предусмотрите конструктора для класса. Здесь он называется конструктором по умолчанию потому, что является конструктором без аргументов, который в действительности переопределяет конструктор по умолчанию компилятора.

Если вы предусмотрите конструктор с аргументами, то система не станет обеспечивать конструктор по умолчанию. Хотя это может показаться запутанным, правило гласит, что конструктор по умолчанию компилятора добавляется, только если вы не предусмотрите в своем коде *никаких* конструкторов.

В этом конструкторе атрибутам `name` и `myCab` присвоено значение `null`:

```
name = null;
myCab = null;
```

ПУСТОЕ ЗНАЧЕНИЕ NULL

Во многих языках программирования значение null представляет собой пустое значение. Это может показаться эзотерикой, однако присвоение атрибутам значения null — рациональная методика программирования. Проверка той или иной переменной на предмет null позволяет выяснить, было ли значение должным образом инициализировано. На-

пример, вам может понадобиться объявить атрибут, который позднее потребует от пользователя ввести данные. Таким образом, вы сможете инициализировать атрибут значением null до того, как пользователю представится возможность ввести данные. Присвоив атрибуту null (что является допустимым условием), позднее вы сможете проверить, было ли задано для него надлежащее значение. Следует отметить, что в некоторых языках программирования нельзя присваивать значение null атрибутам и переменным типа String. Например, при использовании .NET придется указывать name = string.empty;.

Как мы уже знаем, инициализация атрибутов в конструкторах — это всегда хорошая идея. В том же духе правильной методикой программирования будет последующая проверка значения атрибута с целью выяснить, равно ли оно `null`. Благодаря этому вы сможете избежать головной боли в будущем, если для атрибута или объекта будет задано ненадлежащее значение. Например, если вы используете ссылку `myCab` до того, как ей будет присвоен реальный объект, то, скорее всего, возникнет проблема. Если вы зададите для ссылки `myCab` значение `null` в конструкторе, то позднее, когда попытаетесь использовать ее, сможете проверить, по-прежнему ли она имеет значение `null`. Может быть сгенерировано исключение, если вы станете обращаться с неинициализированной ссылкой так, будто она была инициализирована надлежащим образом.

Рассмотрим еще один пример: если у вас имеется класс `Employee`, содержащий атрибут `spouse` (возможно, для страховки), то вам лучше предусмотреть все на случай, если тот или иной работник окажется не состоящим в браке. Изначально присвоив атрибуту значение `null`, вы сможете позднее проверить соответствующий статус.

Второй конструктор дает пользователю класса возможность инициализировать атрибуты `name` и `myCab`:

```
public Cabbie(String iName, String serialNumber) {

  name = iName;
  myCab = new Cab(serialNumber);

}
```

В данном случае пользователь указал бы две строки в списке параметров конструктора для того, чтобы инициализировать атрибуты надлежащим образом. Обратите внимание, что в этом конструкторе создается экземпляр объекта `myCab`:

```
myCab = new Cab(serialNumber);
```

В результате выполнения этой строки кода для объекта `Cab` будет выделена память. На рис. 4.3 показано, как на новый экземпляр объекта `Cab` ссылается атрибут `myCab`. Благодаря применению двух конструкторов в этом примере демонстрируется перегрузка методов. Обратите внимание, что все конструкторы определены как `public`. В этом есть смысл, поскольку в данном случае ясно, что конструкторы являются частью интерфейса класса. Если бы конструкторы были закрытыми, то другие объекты не имели бы к ним доступа и таким образом не смогли бы создавать экземпляры объекта `Cab`.

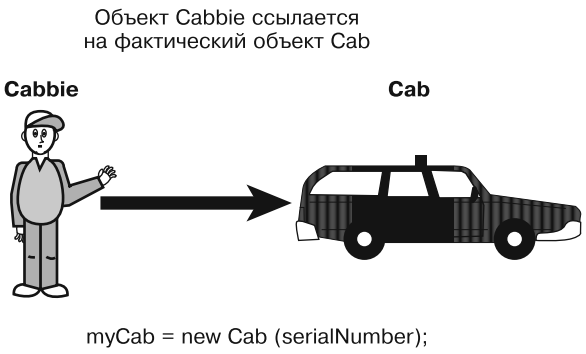

Рис. 4.3. Объект Cabbie, ссылающийся на объект Cab

Методы доступа

В большинстве, если не во всех примерах, приведенных в этой книге, атрибуты определяются как `private`, из-за чего у любых других объектов нет прямого доступа к этим атрибутам. Было бы глупо создавать объект в изоляции, которая не позволит ему взаимодействовать с другими объектами, — мы ведь хотим, чтобы он мог делиться с ними соответствующей информацией. Есть ли необходимость инспектировать и иногда изменять значения атрибутов других классов? Ответом на этот вопрос будет, конечно же, «да». Бывает много ситуаций, когда тому или иному объекту требуется доступ к атрибутам другого объекта; однако это не обязательно должен быть прямой доступ.

Класс должен очень хорошо защищать свои атрибуты. Например, вы не захотите, чтобы у объекта A была возможность инспектировать или изменять значения атрибутов объекта B, если объект B не сможет при этом все контролировать. На это есть несколько причин, и большинство из них сводится к целостности данных и эффективной отладке.

Предположим, что в классе `Cab` есть дефект. Вы отследили проблему, что привело вас к атрибуту `name`. Каким-то образом он перезаписывается, а при выполнении некоторых запросов имен появляется мусор. Если бы `name` был `public` и любой класс мог изменять его значение, то вам пришлось бы просмотреть весь возможный код в попытке найти фрагменты, которые ссылаются на `name` и изменяют его значение. Однако если бы вы разрешили только объекту `Cabbie` изменять значение `name`, то вам пришлось бы выполнять поиск только в классе `Cabbie`. Такой доступ обеспечивается методом особого типа, называемым *методом доступа*. Иногда методы доступа называются геттерами и сеттерами, а порой — просто `get()` и `set()`. По соглашению в этой книге мы указываем методы с префиксами `set` и `get`, как показано далее:

```
// Задание значения для name, относящегося к Cabbie
public void setName(String iName) {
  name = iName;
}

// Извлечение значения name, относящегося к Cabbie
```

```
public String getName() {
  return name;
}
```

В этом фрагменте кода объект `Supervisor` должен отправить запрос объекту `Cabbie` на возврат значения его атрибута `name` (рис. 4.4). Важно здесь то, что объект `Supervisor` сам по себе не сможет извлечь информацию; ему придется запросить сведения у объекта `Cabbie`. Эта концепция важна на многих уровнях. Например, у вас мог бы иметься метод `setAge()`, который проверяет, является ли введенное значение возраста `0` или меньшей величиной. Если значение возраста окажется меньше `0`, то метод `setAge()` может отказаться задавать это некорректное значение. Обычно сеттеры применяются для обеспечения того или иного уровня целостности данных.

Рис. 4.4. Запрос информации

Это также проблема безопасности. Например, у вас имеется требующая защиты информация: пароли или данные расчета заработной платы, доступ к которым вы хотите контролировать. Таким образом, доступ к данным с помощью геттеров и сеттеров обеспечивает возможность использования механизмов вроде проверки паролей и прочих методик валидации. Это значительно укрепляет целостность данных.

Обратите внимание, что метод `getCompanyName` объявлен как `static`, как метод класса; методы классов подробнее рассматриваются в главе 3. Помните, что атрибут `companyName` тоже объявлен как `static`. Метод, как и атрибут, может быть объявлен как `static` для указания на то, что на весь соответствующий класс приходится только одна копия этого метода.

ОБЪЕКТЫ

Вообще говоря, на каждый объект не приходится по одной физической копии каждого нестатического метода. В этом случае каждый объект указывал бы на один и тот же машинный код. Однако на концептуальном уровне вы можете представлять, что объекты полностью независимы и содержат собственные атрибуты и методы.

Приведенный далее фрагмент кода демонстрирует пример определения метода, а на рис. 4.5 показано, как на один и тот же код указывает несколько объектов.

```
// Извлечение значения name, относящегося к Cabbie
public static String getCompanyName() {
  return companyName;
}
```

СТАТИЧЕСКИЕ АТРИБУТЫ

Если атрибут является статическим, а класс обеспечивает для него сеттер, то любой объект, вызывающий этот сеттер, будет изменять единственную копию. Таким образом, значение этого атрибута изменится для всех объектов.

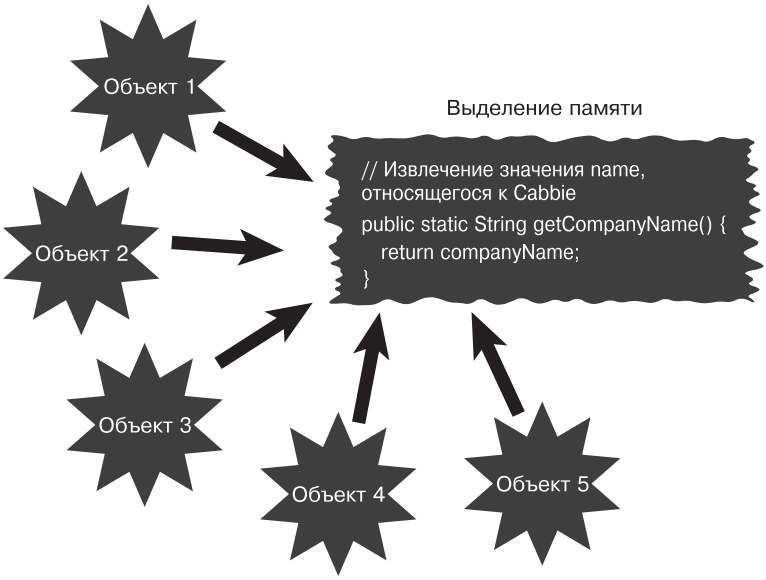

Рис. 4.5. Выделение памяти в случае с методами

Методы открытых интерфейсов

Как конструкторы, так и методы доступа объявляются как `public` и относятся к открытому интерфейсу. Они выделяются в силу своей особой важности для конструкции класса. Однако значительная часть *реальной* работы выполняется в других методах. Как уже отмечалось в главе 2, методы открытых интерфейсов имеют тенденцию быть очень абстрактными, а реализация склонна быть более конкретной. Для нашего класса мы предусмотрим метод `giveDestination`, который будет частью открытого метода и позволит пользователю указать, куда он хочет «отправиться»:

```
public void giveDestination (){

}
```

На данный момент не важно, что содержится внутри этого метода. Главное, что он является открытым методом и частью открытого интерфейса для соответствующего класса.

Методы закрытых реализаций

Несмотря на то что все методы, рассмотренные в этой главе, определяются как `public`, не все методы в классе являются частью открытого интерфейса. Методы в том или ином классе обычно скрыты от других классов и объявляются как `private`:

```
private void turnRight(){
}

private void turnLeft() {
}
```

Эти закрытые методы призваны быть частью реализации, а не открытого интерфейса. У вас может возникнуть вопрос насчет того, кто будет вызывать данные методы, если этого не сможет сделать ни один другой класс. Ответ прост: вы, возможно, уже подозревали, что эти методы можно вызывать изнутри метода `giveDestination`:

```
public void giveDestination (){

  ... код

  turnRight();
  turnLeft();

  ... еще код

}
```

В качестве еще одного примера можно привести возможную ситуацию, когда у вас имеется внутренний метод, обеспечивающий шифрование, который вы будете вызывать изнутри самого класса. Коротко говоря, этот метод шифрования нельзя вызвать извне созданного экземпляра объекта как такового.

Главное здесь состоит в том, что закрытые методы являются строго частью реализации и недоступны другим классам.

Резюме

В этой главе мы заглянули внутрь класса и рассмотрели фундаментальные концепции, необходимые для понимания принципов создания классов. Хотя в этой главе был использован скорее практический подход, в главе 5 классы будут рассмотрены с общей точки зрения проектировщика.

Ссылки

- *Фаулер Мартин*. UML. Основы (UML Distilled). 3-е изд. — Бостон: Addison-Wesley Professional, 2003.

- *Гилберт Стивен и Маккарти Билл.* Объектно-ориентированное проектирование на Java (Object-Oriented Design in Java). — Беркли: The Waite Group Press (Pearson Education), 1998.
- *Тима Пол, Торок Габриэл и Даунинг Трой.* Учебник по Java для начинающих (Java Primer Plus). — Беркли: The Waite Group, 1996.

Примеры кода, использованного в этой главе

Приведенный далее код написан на C# .NET. Эти примеры соответствуют Java-коду, продемонстрированному в текущей главе.

Пример TestCab: C# .NET

```
using System;

namespace ConsoleApplication1
{
  class TestPerson
  {
    public static void Main()
    {
      Cabbie joe = new Cabbie("Joe", "1234");

      Console.WriteLine(joe.Name);
      Console.ReadLine();
    }
  }

  public class Cabbie
  {

    private string _Name;
    private Cab _Cab;

    public Cabbie() {

      _Name = null;
      _Cab = null;

    }
    public Cabbie(string name, string serialNumber) {

      _Name = name;
      _Cab = new Cab(serialNumber);

    }

    // Методы
```

```
    public String Name
    {
      get { return _Name; }
      set { _Name = value; }
    }

  }

  public class Cab
  {

    public Cab (string serialNumber) {

      SerialNumber = serialNumber;

    }

    // Это свойство открыто для get, но закрыто для set
    public string SerialNumber { get; private set; }

  }
}
```

Глава 5
Руководство по проектированию классов

Как уже отмечалось, объектно-ориентированное программирование поддерживает идею создания классов в виде полных пакетов, которые инкапсулируют данные и поведения единого целого. Таким образом, класс должен представлять логический компонент, например такси.

В этой главе приведены рекомендации по проектированию качественных классов. Ясно, что список вроде этого нельзя считать исчерпывающим. Вы, несомненно, добавите большое количество указаний в свой личный перечень, также включив в него полезные инструкции от других разработчиков.

Моделирование реальных систем

Одна из основных целей объектно-ориентированного программирования — моделирование реальных систем способами, которые схожи с фактическим образом мышления людей. Проектирование классов — это объектно-ориентированный вариант создания таких моделей. Вместо использования структурного, или нисходящего, подхода, при котором данные и поведения — это логически раздельные сущности, объектно-ориентированный подход инкапсулирует данные и поведения в объектах, взаимодействующих друг с другом. Мы больше не представляем себе проблему как последовательность событий или программ, воздействующих на отдельные файлы данных. Элегантность этого подхода состоит в том, что классы буквально моделируют реальные объекты и их взаимодействие с другими реальными объектами.

Эти взаимодействия происходят почти так же, как взаимодействия между реальными объектами, такими, например, как люди. Поэтому при создании классов вам следует проектировать их тем способом, который позволит представить истинное поведение объекта. Воспользуемся примером с `Cabbie` из предыдущих глав. Классы `Cab` и `Cabbie` моделируют реальную сущность. Как показано на рис. 5.1, объекты `Cab` и `Cabbie` инкапсулируют свои данные и поведения, а также взаимодействуют с помощью открытых интерфейсов друг друга.

При переходе на объектно-ориентированную разработку в первый раз многие люди по-прежнему мыслят структурным образом. Одна из главных ошибок — создание класса, который обладает поведениями, но не имеет своих данных. Это не то, что вам нужно, поскольку при этом не используются преимущества инкапсуляции.

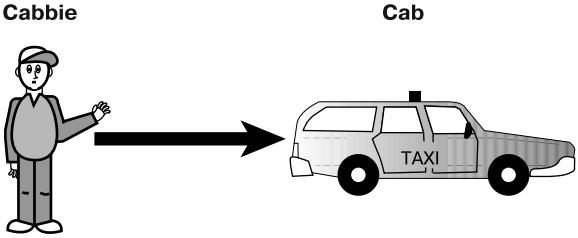

Рис. 5.1. Cabbie и Cab — реальные объекты

ПРИМЕЧАНИЕ

Одной из моих любимых книг, содержащих указания и рекомендации по проектированию, остается та, что написана Скоттом Майерсом и называется «Эффективное использование C++: 50 рекомендаций по улучшению ваших программ и проектов» (Effective C++: 50 Specific Ways to Improve Your Programs and Designs). В ней важная информация о проектировании программ преподносится очень лаконично.

Одна из причин, по которым книга «Эффективное использование C++» так сильно интересует меня, заключается в том, что, поскольку C++ обратно совместим с языком программирования C, компилятор позволит вам писать структурированный код на C++ без применения принципов объектно-ориентированного проектирования. Именно по этой причине так важно следовать указаниям вроде тех, что приведены в упомянутой книге. Как я уже отмечал ранее, при собеседовании некоторые люди утверждают, что занимаются объектно-ориентированным программированием, просто потому, что они пишут код на C++. Это свидетельствует о полном непонимании того, что такое объектно-ориентированное проектирование. По этой причине вам, возможно, стоит уделять больше внимания задачам объектно-ориентированного проектирования при использовании таких языков, как C++ и Objective-C, а не Java или .NET.

Определение открытых интерфейсов

К настоящему времени должно быть понятно, что, пожалуй, наиболее важная задача при проектировании класса — обеспечение минимального открытого интерфейса. Создание класса полностью сосредоточено на обеспечении чего-то полезного и компактного. В своей книге «Объектно-ориентированное проектирование на Java» *(Object-Oriented Design in Java)* Гилберт и Маккарти пишут, что «интерфейс хорошо спроектированного объекта описывает услуги, оказание которых требуется клиенту». Если класс не будет предоставлять полезных услуг пользователям, то его вообще не следует создавать.

Минимальный открытый интерфейс

Обеспечение минимального открытого интерфейса позволяет сделать класс как можно более компактным. Цель состоит в том, чтобы предоставить пользователю именно тот интерфейс, который даст ему возможность правильно выполнить

соответствующую работу. Если открытый интерфейс окажется неполным (то есть будет отсутствовать поведение), то пользователь не сможет сделать всю работу. Если для открытого интерфейса не будет предусмотрено соответствующих ограничений (то есть пользователю предоставят доступ к поведению, что будет излишним или даже опасным), то в результате может возникнуть необходимость отладки или, возможно, даже появятся проблемы с целостностью и защитой системы.

Создание класса — серьезная задача, и, как и на всех этапах процесса проектирования, очень важно, чтобы пользователи были вовлечены в него с самого начала и на всем протяжении стадии тестирования. Таким образом, это позволит создать полезный класс, а также обеспечить надлежащие интерфейсы.

РАСШИРЕНИЕ ИНТЕРФЕЙСА

Даже если открытого интерфейса класса окажется недостаточно для определенного приложения, объектная технология с легкостью позволит расширить и адаптировать этот интерфейс. Коротко говоря, если спроектировать новый класс с учетом наследования и композиции, то он сможет использовать существующий класс и создать новый класс с расширенным интерфейсом.

Чтобы проиллюстрировать это, снова обратимся к примеру с `Cabbie`. Если другим объектам в системе потребуется извлечь значение `name` объекта `Cabbie`, то класс `Cabbie` должен будет обеспечивать открытый интерфейс для возврата этого значения; им будет метод `getName()`. Таким образом, если объекту `Supervisor` понадобится извлечь значение `name` объекта `Cabbie`, то ему придется вызвать метод `getName()` из объекта `Cabbie`. Фактически `Supervisor` будет запрашивать у `Cabbie` значение его `name` (рис. 5.2).

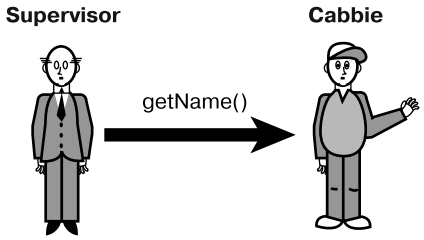

Рис. 5.2. Открытый интерфейс определяет, как объекты взаимодействуют

Пользователи вашего кода не должны ничего знать о его внутренней работе. Им нужно знать лишь то, как создавать экземпляры и использовать объекты. Иными словами, обеспечивайте для пользователей способ выполнять требуемые им действия, но скрывайте детали.

Скрытие реализации

Необходимость скрывать реализацию уже рассматривалась очень подробно. Хотя определение открытого интерфейса — это задача проектирования, «вращающаяся» вокруг пользователей класса, реализация не должна их вообще касаться. Реализа-

ция будет предоставлять услуги, необходимые пользователям, однако способ их предоставления не следует делать очевидным для пользователей. Класс окажется наиболее полезен, если реализация сможет изменяться, никак не затрагивая пользователей. Например, изменение реализации не должно неизбежно влечь за собой изменение в пользовательском программном коде.

КЛИЕНТ В ПРОТИВОПОСТАВЛЕНИИ С ПОЛЬЗОВАТЕЛЕМ

Иногда я использую термин «клиент» вместо «пользователь», когда говорю о людях, которые в действительности будут использовать программное обеспечение. Поскольку я консультант, то пользователи системы фактически будут клиентами. В том же духе пользователи, относящиеся к вашей организации, будут называться внутренними клиентами. Это может показаться тривиальным, но я думаю, что важно считать всех конечных пользователей фактическими клиентами и вы должны удовлетворить их требования.

В примере с `Cabbie` класс с аналогичным названием мог содержать поведение, касающееся того, как таксист завтракает. Однако объекту `Supervisor` не нужно знать, что таксист, представляемый объектом `Cabbie`, ел на завтрак. Таким образом, это поведение является частью реализации объекта `Cabbie` и не должно быть доступно другим объектам в этой системе (рис. 5.3). Гилберт и Маккарти пишут, что основная директива инкапсуляции заключается в том, что «все поля будут закрытыми». Таким образом, ни одно из полей в классе не будет доступно из других объектов.

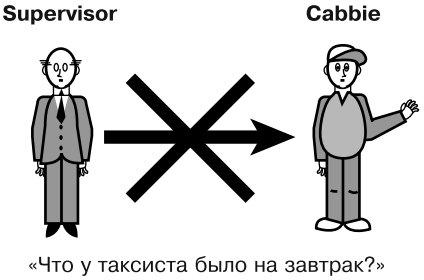

Рис. 5.3. Объектам не нужно знать некоторые детали реализации

Проектирование надежных конструкторов (и, возможно, деструкторов)

При проектировании класса одна из самых важных соответствующих задач — принять решение о том, как этот класс будет сконструирован. Конструкторы были рассмотрены в главе 3. Загляните в нее снова, если вам потребуется освежить свои знания насчет основных принципов проектирования конструкторов.

Прежде всего конструктор должен задать для объекта его начальное, надежное состояние. Сюда входит выполнение таких задач, как инициализация атрибутов и управление памятью. Вам также потребуется убедиться в том, что объект должным образом сконструирован в состоянии по умолчанию. Как вариант, можно обеспечить конструктор для обработки этой стандартной ситуации.

При использовании языков программирования, в которых имеются деструкторы, жизненно важно, чтобы эти деструкторы включали соответствующие функции очистки. В большинстве случаев такая очистка связана с высвобождением системной памяти, полученной объектом в какой-то момент. Java и .NET автоматически регенерируют память с помощью механизма сборки мусора. При использовании языков программирования вроде C++ разработчик должен включать код в деструктор для надлежащего высвобождения памяти, которую объект занимал во время своего существования. Если проигнорировать эту функцию, то в результате произойдет утечка памяти.

УТЕЧКИ ПАМЯТИ

Когда объект не высвобождает надлежащим образом память, которую занимал во время своего жизненного цикла, она оказывается утраченной для всей операционной системы до тех пор, пока выполняется приложение, создавшее этот объект. Допустим, множественные объекты одного и того же класса создаются, а затем уничтожаются, возможно, в рамках некоторого цикла. Если эти объекты не высвободят свою память, когда окажутся вне области видимости, то соответствующая утечка памяти приведет к исчерпанию доступного пула системной памяти. В какой-то момент может оказаться израсходовано столько памяти, что у системы не останется свободного ее объема для выделения. Это означает, что любое приложение, выполняющееся в системе, не сумеет получить хоть сколько-нибудь памяти. Это может ввергнуть приложение в нестабильное состояние и даже привести к блокировке системы.

Внедрение обработки ошибок в класс

Как и при проектировании конструкторов, жизненно важно продумать, как класс будет обрабатывать ошибки. Обработка ошибок была подробно рассмотрена в главе 3.

Почти наверняка можно сказать, что каждая система будет сталкиваться с непредвиденными проблемами. Поэтому не стоит игнорировать потенциальные ошибки. Разработчик хорошего класса (или любого кода, если на то пошло) предвидит потенциальные ошибки и предусматривает код для обработки таких ситуаций, когда они имеют место.

Согласно общему правилу приложение никогда не должно завершаться аварийно. При обнаружении ошибки система должна либо «починить» себя и продолжить функционировать, либо корректно завершить свою работу без потери каких-либо данных, важных для пользователя.

Документирование класса и использование комментариев

Тема комментариев и документирования поднимается в каждой книге и статье по программированию, при каждой инспекции кода, в каждой дискуссии насчет грамотного подхода к проектированию, в которой вы участвуете. К сожалению, комментарии и надлежащее документирование зачастую не принимаются всерьез или, что еще хуже, игнорируются.

Большинству разработчиков известно, что следует тщательно документировать свой код, однако обычно они не желают тратить на это время. Но грамотный подход

к проектированию практически невозможен без правильных методик документирования. На уровне класса область видимости может оказаться достаточно небольшой для того, чтобы разработчик смог отделаться низкосортной документацией. Однако, когда класс передается кому-то другому для расширения и/или сопровождения либо становится частью более крупной системы (что и должно случиться), отсутствие надлежащей документации и комментариев может подорвать всю систему.

Многие люди уже говорили все это раньше. Один из самых важных аспектов грамотного подхода к проектированию, будь то проектирование класса или чего-то другого, — тщательное документирование процесса. Такие реализации, как Java и .NET, предусматривают специальный синтаксис комментариев для облегчения процесса документирования. Загляните в главу 4, чтобы увидеть соответствующий синтаксис.

СЛИШКОМ БОЛЬШОЕ КОЛИЧЕСТВО ДОКУМЕНТАЦИИ

Имейте в виду, что чрезмерное комментирование тоже может привести к проблемам. Слишком большое количество документации и/или комментариев может мешать и вообще действовать во вред целям документирования. Точно так же, как и при грамотном подходе к проектированию классов, делайте так, чтобы документация и комментарии были понятными и в них все говорилось по существу.

Создание объектов с прицелом на взаимодействие

Мы можем с уверенностью сказать, что почти ни один класс не существует в изоляции. В большинстве случаев нет никаких причин создавать класс, если он не будет взаимодействовать с другими классами. Это факт в «жизни» класса. Класс станет оказывать услуги другим классам, запрашивать услуги других классов либо делать и то и другое. В последующих главах мы рассмотрим различные способы, посредством которых классы могут взаимодействовать друг с другом.

В приводившемся ранее примере `Cabbie` и `Supervisor` не являются автономными сущностями; они взаимодействуют друг с другом на разных уровнях (рис. 5.4).

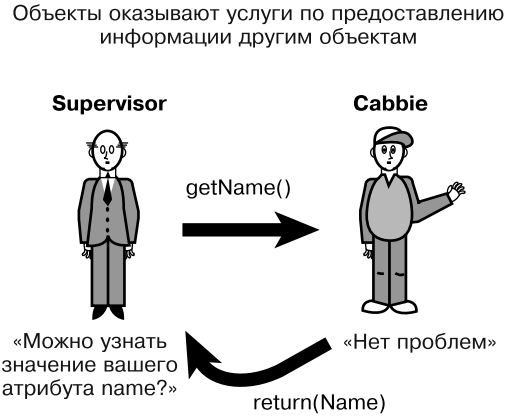

Рис. 5.4. Объекты должны запрашивать информацию

При проектировании класса убедитесь, что вы отдаете себе отчет в том, как другие объекты будут взаимодействовать с ним.

Проектирование с учетом повторного использования

Объекты могут повторно использоваться в разных системах, а код следует писать с учетом такого повторного использования. Например, когда класс `Cabbie` разработан и протестирован, его можно применять везде, где требуется такой класс. Чтобы сделать класс пригодным к использованию в разных системах, его нужно проектировать с учетом повторного использования. Именно здесь потребуется хорошо поразмыслить в процессе проектирования. Попытка предсказать все возможные сценарии, при которых объект `Cabbie` должен будет работать, — непростая задача. Более того, это фактически невозможно сделать.

Проектирование с учетом расширяемости

Добавление новых функций в класс может быть таким же простым, как расширение существующего класса, добавление нескольких новых методов и модификация поведений других. Нет надобности все переписывать. Именно здесь в дело вступает наследование. Если вы только что написали класс `Person`, то должны принимать во внимание тот факт, что позднее вам, возможно, потребуется написать класс `Employee` или `Vendor`. Поэтому лучше всего сделать так, чтобы `Employee` наследовал от `Person`; в этом случае класс `Person` будет называться *расширяемым*. Вы не захотите проектировать класс `Person` таким образом, чтобы он содержал поведение, которое будет препятствовать его расширяемости другими классами, например `Employee` или `Vendor` (предполагается, что при проектировании вы будете действительно нацелены на то, чтобы другие классы расширяли `Person`). Например, вы не захотели бы включать в класс `Employee` функциональность, характерную для супервизорных функций. Если бы вы все же решились на это, а класс, которому не требуется такая функциональность, наследовал бы от `Employee`, то у вас возникла бы проблема.

Этот аспект затрагивает рекомендации по абстрагированию, о которых шла речь ранее. `Person` должен содержать только данные и поведения, специфичные для него. Другие классы тогда смогут быть его подклассами и наследовать соответствующие данные и поведения.

КАКИЕ АТРИБУТЫ И МЕТОДЫ МОГУТ БЫТЬ СТАТИЧЕСКИМИ?

Важно определиться с тем, какие атрибуты и методы могут быть объявлены как static. Снова загляните в главу 3, где рассматривалось использование ключевого слова static. Вы сможете понять, как включать в свои классы статические атрибуты и методы, — они совместно используются всеми объектами того или иного класса.

Делаем имена описательными

Ранее мы рассмотрели использование надлежащей документации и комментариев. Следование соглашению об именовании классов, атрибутов и методов является

схожей темой. Существует большое количество соглашений об именовании, а то, какое именно из них вы выберете, не так важно, как просто выбор одного соглашения и следование ему. Однако при выборе соглашения убедитесь в том, что, придумывая имена для классов, атрибутов и методов, вы не только следуете соответствующему соглашению, но и делаете эти имена описательными. Когда кто-нибудь прочтет одно из этих имен, он должен будет способен, исходя из имени, сказать, что представляет собой соответствующий объект. Какие конкретно соглашения об именовании будут использоваться, в различных организациях зачастую обуславливается стандартами программирования.

ПРАВИЛЬНОЕ СОГЛАШЕНИЕ ОБ ИМЕНОВАНИИ

Убедитесь в том, что соглашение об именовании имеет смысл. Люди часто перебарщивают и придумывают соглашения, которые имеют смысл для них, но оказываются совершенно непонятными для других людей. Будьте осторожны, вынуждая других следовать тому или иному соглашению. Удостоверьтесь в том, что соглашения разумны и всем заинтересованным людям понятен смысл, который в них кроется.

Делая имена описательными, вы будете придерживаться правильной методики разработки, которая выходит за пределы различных парадигм разработки.

Абстрагирование непереносимого кода

Если вы проектируете систему, которая должна задействовать непереносимый (нативный) код (то есть код, который будет выполняться только на определенной аппаратной платформе), то вам придется абстрагировать его от соответствующего класса. Под абстрагированием мы подразумеваем изоляцию непереносимого кода в его собственном классе или по крайней мере в его собственном методе (который может быть переопределен). Например, если вы пишете код для доступа к последовательному порту определенного аппаратного обеспечения, то вам следует создать класс-обертку для работы с ним. Ваш класс затем должен будет отправить сообщение классу-обертке для получения информации и услуг, которые ему нужны. Не размещайте зависимый от системы код в своем первичном классе (рис. 5.5).

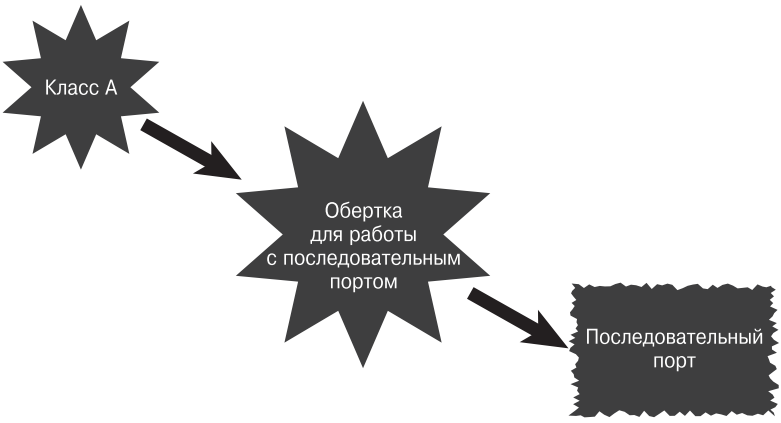

Рис. 5.5. Обертка для работы с последовательным портом

Рассмотрим, например, ситуацию, когда программист взаимодействует непосредственно с аппаратным обеспечением. В таких случаях объектный код разных платформ, скорее всего, будет сильно отличаться, поэтому код потребуется написать для каждой платформы. Однако если функциональность будет заключена в класс-обертку, то пользователь класса сможет взаимодействовать непосредственно с оберткой и ему не придется беспокоиться о различном низкоуровневом коде. Класс-обертка разберется в различиях платформ и решит, какой код вызывать.

Обеспечение возможности осуществлять копирование и сравнение

В главе 3 мы говорили о копировании и сравнении объектов. Важно понимать, как осуществляются эти действия. Вы, возможно, не захотите или не будете рассчитывать на простую побитовую копию или операцию сравнения. Вы должны убедиться в том, что ваш класс ведет себя так, как от него ожидается, а это означает, что вам придется потратить некоторое время на проектирование способов копирования и сравнения объектов.

Сведение области видимости к минимуму

Сведение области видимости к минимуму идет рука об руку с абстрагированием и скрытием реализации. Идея заключается в том, чтобы локализовать атрибуты и поведения настолько, насколько это представляется возможным. Таким образом, сопровождать, тестировать и расширять классы станет намного легче.

ОБЛАСТЬ ВИДИМОСТИ И ГЛОБАЛЬНЫЕ ДАННЫЕ

Минимизация области видимости глобальных переменных — хороший стиль программирования, но она нехарактерна для объектно-ориентированного программирования. При структурной разработке допускается использование глобальных переменных, однако это рискованно.

Фактически в сфере объектно-ориентированной разработки нет глобальных данных. Статические атрибуты и методы совместно используются объектами одного и того же класса, однако они недоступны остальным объектам.

Например, если у вас имеется метод, которому требуется временный атрибут, пусть он будет локальным. Взгляните на приведенный далее код:

```
public class Math {

  int temp=0;

  public int swap (int a, int b) {

    temp = a;
    a=b;
    b=temp;
```

```
    return temp;

  }

}
```

Что не так с этим классом? Проблема заключается в том, что необходимо, чтобы атрибут `temp` был только в рамках области видимости метода `swap()`. Нет никаких причин для того, чтобы он был на уровне класса. Таким образом, вам следует переместить `temp` в область видимости метода `swap()`:

```
public class Math {

  public int swap (int a, int b) {

    int temp=0;

    temp = a;
    a=b;
    b=temp;

    return temp;

  }

}
```

Вот что подразумевается под сведением области видимости к минимуму.

Класс должен отвечать за себя

В подготовительном курсе на основе своей книги «Учебник по Java для начинающих» *(Java Primer Plus)* Тима, Торок и Даунинг приводят требование к проектированию классов, согласно которому все объекты должны отвечать за выполнение своих действий во всех возможных случаях. Рассмотрим попытку вывести круг.

Для иллюстрирования прибегнем к примеру, который не является объектно-ориентированным. В этом примере команде `print` в качестве аргумента передается `circle`, после чего выводится круг (рис. 5.6):

```
print(circle);
```

Функциям `print`, `draw` и др. необходим оператор (или что-то вроде конструкции `if/else`) для выяснения того, что делать с определенной фигурой, обозначенной с помощью переданного им параметра. В данной ситуации можно вызвать отдельную программу печати для каждой фигуры:

```
printCircle(circle);
printSquare(square);
```

Каждый раз при добавлении новой фигуры вам потребуется добавить соответствующий параметр в операторы `case` всех функций:

```
switch (shape) {
  case 1: printCircle(circle); break;
  case 2: printSquare(square); break;
  case 3: printTriangle(triangle); break;
  default: System.out.println("Недопустимая фигура.");break;
}
```

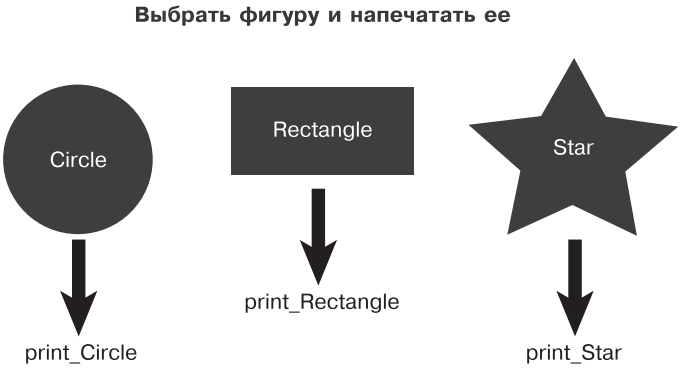

Рис. 5.6. Не являющийся объектно-ориентированным пример сценария, при котором используется print

Рассмотрим объектно-ориентированный пример. Используя полиморфизм и относя `Circle` к соответствующей категории, `Shape` выясняет, что речь идет о круге, и знает, как вывести требуемую фигуру (рис. 5.7).

```
shape.print(); // Фигурой в данном случае является круг
shape.print(); // Фигурой в данном случае является квадрат
```

Здесь важно понимать, что вызов остается тем же; контекст фигуры обуславливает реакцию системы.

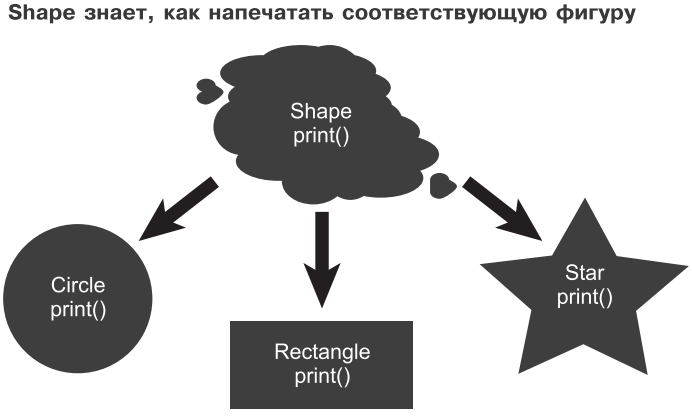

Рис. 5.7. Объектно-ориентированный пример сценария, при котором используется print

Проектирование с учетом сопровождаемости

Проектирование практичных и компактных классов обеспечивает высокий уровень сопровождаемости. Точно так же, как вы проектируете классы с учетом расширяемости, вам следует проектировать их с учетом будущего сопровождения.

Процесс проектирования классов вынудит вас разделять ваш код на множество идеально управляемых фрагментов. Отдельные фрагменты кода гораздо удобнее в сопровождении, чем его более крупные фрагменты (по крайней мере так считается). Один из наилучших способов обеспечить сопровождаемость — уменьшить количество взаимозависимого кода, то есть изменения в одном классе не должны сказываться, даже минимально, на других классах.

ТЕСНО СВЯЗАННЫЕ КЛАССЫ

Классы, которые сильно зависят друг от друга, считаются тесно связанными. Таким образом, если изменение, внесенное в один класс, приводит к изменению в другом классе, то эти два класса будут считаться тесно связанными. Классы, лишенные таких зависимостей, обладают очень низкой степенью связанности. Более подробно об этом вы сможете узнать из книги Скотта Амблера (Scott Ambler) «Введение в объектно-ориентированную технологию» (The Object Primer).

Если классы изначально правильно спроектированы, то любые изменения в системе должны вноситься только в реализацию объекта. Изменений открытого интерфейса следует избегать любой ценой. Любые изменения открытого интерфейса приведут к волновым эффектам во всех системах, задействующих этот интерфейс.

Например, если внести изменение в метод `getName()` класса `Cabbie`, то все места во всех системах, где используется этот интерфейс, потребуется изменить и перекомпилировать. Обнаружение всех соответствующих вызовов методов — это грандиозная задача, а вероятность упустить один из них довольно высока.

Для обеспечения высокого уровня сопровождаемости делайте так, чтобы степень связанности ваших классов была как можно ниже.

Использование итерации в процессе разработки

Как и в большинстве функций проектирования и программирования, рекомендуется использовать итеративный процесс. Это хорошо согласуется с концепцией обеспечения минимальных интерфейсов. По сути это означает, что *не нужно писать сразу весь код*! Делайте это небольшими шагами, создавая и тестируя код на каждом этапе. Хороший план тестирования позволит быстро выявить все области, где обеспечены недостаточные интерфейсы. Таким образом, процесс можно будет повторять до тех пор, пока у класса не появятся надлежащие интерфейсы. Этот процесс тестирования затрагивает не только написанный код. Очень полезно протестировать то, что вы спроектировали, с применением критического анализа и других методик оценки результатов. Использование итеративных процессов облегчает жизнь тестировщикам, поскольку они вовлекаются в ход событий еще на раннем этапе, а не просто получают в свои руки систему, которую им «перебрасывают через стену» в конце процесса разработки.

Тестирование интерфейса

Минимальные реализации интерфейса часто называются *заглушками* (Гилберт и Маккарти хорошо рассмотрели тему заглушек в своей книге под названием «Объектно-ориентированное проектирование на Java» *(Object-Oriented Design in Java)*. Используя заглушки, вы сможете тестировать интерфейсы без написания *реального* кода. В приведенном далее примере вместо подключения к настоящей базе данных заглушки применяются для проверки того, что интерфейсы работают правильно (с точки зрения пользователя, ведь интерфейсы предназначены для них). Таким образом, на данном этапе нет необходимости в реализации. Более того, обеспечение завершенной реализации на данном этапе может стоить драгоценного времени и сил, поскольку конструкция интерфейса повлияет на реализацию, а интерфейс при этом еще не будет завершен.

Обратите внимание на рис. 5.8: когда пользовательский класс отправляет сообщение классу `DataBaseReader`, информация, возвращаемая пользовательскому классу, предоставляется кодовыми заглушками, а не настоящей базой данных (фактически базы данных, скорее всего, даже не существует). Когда интерфейс окажется завершен, а реализация будет находиться в процессе разработки, можно будет подключиться к базе данных, а заглушки — отключить.

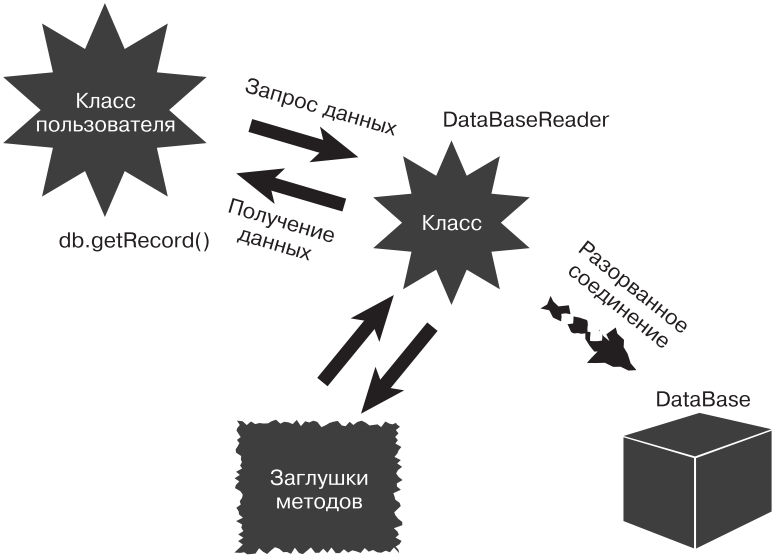

Рис. 5.8. Применение заглушек

Вот пример кода, который задействует внутренний массив для имитации работающей базы данных (хотя и простой):

```
public class DataBaseReader {

  private String db[] = { "Record1",
    "Record2",
```

```
      "Record3",
      "Record4",
      "Record5"};

    private boolean DBOpen = false;
    private int pos;

    public void open(String Name){
      DBOpen = true;
    }
    public void close(){
      DBOpen = false;
    }
    public void goToFirst(){
      pos = 0;
    }
    public void goToLast(){
      pos = 4;
    }
    public int howManyRecords(){
      int numOfRecords = 5;

      return numOfRecords;
    }
    public String getRecord(int key){

      /* Специфичная для базы данных реализация */
      return db[key];
    }
    public String getNextRecord(){

      /* Специфичная для базы данных реализация */
      return db[pos++];
    }
  }
```

Обратите внимание на то, как методы имитируют вызовы базы данных. Строки в массиве представляют записи, которые будут сохранены в базу данных. Когда база данных будет успешно интегрирована с системой, она станет использоваться вместо массива.

СОХРАНЕНИЕ ЗАГЛУШЕК

Когда заглушки сделают свое дело, не удаляйте их. Сохраните их в коде для возможного применения в будущем — только позаботьтесь о том, чтобы пользователи не смогли увидеть их. Фактически в той или иной хорошо спроектированной программе, которую вы тестируете, заглушки должны быть интегрированы с конструкцией и сохраняться в программе для более позднего использования. Коротко говоря, встраивайте функциональность для осуществления тестирования прямо в класс!

Сталкиваясь с проблемами при проектировании интерфейсов, вносите изменения и повторяйте процесс до тех пор, пока результат вас не устроит.

Использование постоянства объектов

Постоянство объектов — еще один вопрос, который требуется решать во многих объектно-ориентированных системах. *Постоянство* — это концепция сохранения состояния объекта. Если вы не сохраните объект тем или иным путем при выполнении программы, то он «умрет» и его нельзя будет «воскресить» когда-либо. Такие временные объекты могут работать в некоторых приложениях, однако в большинстве бизнес-систем состояние объекта должно сохраняться для более позднего использования.

ПОСТОЯНСТВО ОБЪЕКТОВ

Несмотря на то что тема постоянства объектов может и не показаться действительно относящейся к руководству по проектированию, я считаю, что обязательно нужно уделить ей внимание при проектировании классов. Я представляю ее здесь, чтобы подчеркнуть, что о постоянстве объектов следует подумать еще на раннем этапе проектирования классов.

В своей простейшей форме объект может сохраняться, будучи сериализованным и записанным в плоский файл. Самая современная технология сейчас базируется на XML. Теоретически объект может сохраняться в памяти, пока не будет уничтожен, но мы сосредоточимся на сохранении постоянных объектов на чем-то вроде запоминающего устройства. Следует учитывать три первичных «запоминающих устройства».

- **Система плоских файлов** — вы можете сохранить объект в плоском файле, сериализовав его. Такой подход имеет очень ограниченное применение.
- **Реляционная база данных** — для преобразования объекта в реляционную модель потребуется некоторое промежуточное программное обеспечение.
- **Объектно-ориентированная база данных** — может оказаться наиболее эффективным способом сделать объекты постоянными, однако вся информация большинства компаний содержится в унаследованных системах, и на данный момент маловероятно, что они решат преобразовать свои реляционные базы данных в объектно-ориентированные.

Сериализация и маршалинг объектов. Мы уже рассматривали проблему использования объектов в средах, которые изначально предназначены для структурного программирования. Хороший образец — пример с промежуточным программным обеспечением, в котором мы записывали объекты в реляционную базу данных. Мы также затронули проблему записи объекта в плоский файл или передачи его по сети.

Для передачи объекта по сети (например, файлу) система должна деконструировать этот объект (сделать его плоским), передать его по сети, а затем реконструировать на другом конце сети. Этот процесс называется *сериализацией* объекта. Действие по передаче объекта по сети называется *маршалингом* объекта. В теории сериализованный объект может быть записан в плоский файл и извлечен позднее в том же состоянии, в котором был записан.

Основной вопрос здесь состоит в том, что при сериализации и десериализации должны использоваться одни и те же спецификации. Это что-то вроде алгоритма

шифрования. Если один объект зашифрует строку, то другому объекту, который захочет расшифровать ее, придется использовать тот же алгоритм шифрования. В Java предусмотрен интерфейс `Serializable`, который обеспечивает соответствующее преобразование.

В C# .NET и Visual Basic .NET имеется интерфейс `ISerializable`, который в документации Microsoft описывается так: *«Позволяет объекту контролировать свою сериализацию и десериализацию»*. Все классы, подлежащие сериализации, должны реализовывать этот интерфейс. Синтаксис как для C# .NET, так и для Visual Basic .NET, приведен далее:

```
' Visual Basic .NET
Public Interface ISerializable

// C# .NET
public interface ISerializable
```

Одна из проблем с сериализацией заключается в том, что этот процесс зачастую подразумевает использование проприетарного решения. Применение XML, о котором мы подробно поговорим позднее, предполагает применение непроприетарной технологии.

Резюме

Эта глава содержит большое количество указаний, которые могут помочь вам в проектировании классов. Однако это ни в коей мере не исчерпывающий список. Вы, несомненно, узнаете и о других правилах, путешествуя по миру объектно-ориентированного проектирования.

В этой главе рассказано о решении задач проектирования касательно отдельных классов. Однако мы уже видели, что тот или иной класс не существует в изоляции. Классы призваны взаимодействовать с другими классами. Группа классов, взаимодействующих друг с другом, представляет собой часть системы. В конечном счете именно такие системы ценны для конечных пользователей. В главе 6 мы рассмотрим тему проектирования полных систем.

Ссылки

- *Майерс Скотт*. Эффективное использование C++ (Effective C++). 3-е изд. — Бостон: Addison-Wesley Professional, 2005.
- *Амблер Скотт*. Введение в объектно-ориентированную технологию (The Object Primer). 3-е изд. — Кембридж: Cambridge University Press, 2004.
- *Яворски Джейми*. Платформа Java 2 в действии (Java 2 Platform Unleashed). — Индианаполис: Sams Publishing, 1999.
- *Гилберт Стивен и Маккарти Билл*. Объектно-ориентированное проектирование на Java (Object-Oriented Design in Java). — Беркли: The Waite Group Press (Pearson Education), 1998.

- *Тима Пол, Торок Габриэл и Даунинг Трой.* Учебник по Java для начинающих (Java Primer Plus). — Беркли: The Waite Group, 1996.
- *Яворски Джейми.* Руководство разработчика на Java 1.1 (Java 1.1 Developers Guide). — Индианаполис: Sams Publishing, 1997.

Примеры кода, использованного в этой главе

Приведенный далее код написан на C# .NET. Эти примеры соответствуют Java-коду, продемонстрированному в текущей главе.

Пример TestMath: C# .NET

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace TestMath
{
  public class Math
  {
    public int swap(int a, int b)
    {

      int temp = 0;

      temp = a;
      a = b;
      b = temp;

      return temp;

    }

  }
  class TestMath
  {
    public static void Main()
    {
      Math myMath = new Math();
      myMath.swap(2, 3);
    }
  }
}
```

Глава 6
Проектирование с использованием объектов

При использовании того или иного программного продукта вы ожидаете, что он будет функционировать так, как это заявлено. К сожалению, не все продукты оправдывают ожидания. Проблема заключается в том, что при создании большого количества продуктов много времени и сил тратится на этапе программирования, а не на стадии проектирования.

Объектно-ориентированное проектирование рекламировалось как надежный и гибкий подход к разработке программного обеспечения. По правде говоря, проектируя объектно-ориентированным способом, вы можете получить как хорошие, так и плохие результаты с той же легкостью, как и при использовании любого другого подхода. Пусть вашу бдительность не притупляет ложное чувство безопасности, основанное лишь на том, что вы применяете самую современную методологию проектирования. Вы должны уделять внимание общей конструкции и тратить достаточное количество времени и сил на создание наилучшего продукта, который только возможен.

В главе 5 мы сконцентрировались на проектировании хороших классов, а в этой главе сосредоточимся на проектировании хороших систем. *Систему* можно определить в виде классов, взаимодействующих друг с другом. Соответствующие методики проектирования развивались на протяжении всей истории разработки программного обеспечения, и теперь вы можете смело пользоваться преимуществами того, что было достигнуто ценой крови, пота и слез ваших предшественников в сфере создания программного обеспечения, независимо от того, применяли они объектно-ориентированные технологии или нет. Во многих случаях вы просто будете брать код, который, возможно, нормально работает уже много лет, и буквально обертывать им свои объекты. Об *обертывании* мы поговорим позднее в этой главе.

Руководство по проектированию

Ошибочно полагать, что может быть одна истинная методология проектирования. На самом деле это, конечно же, не так. Нет правильного или неправильного способа проектирования. Сегодня доступно много методологий, и у каждой из них имеются свои сторонники. Однако главный вопрос состоит не в том, какую методику проектирования использовать, а в том, применять ли ту или иную методику вообще. Все это можно вывести за пределы проектирования, чтобы охватить весь

процесс разработки программного обеспечения. Многие организации не соблюдают стандартного процесса разработки программного обеспечения либо выбирают какой-то один, но не придерживаются его твердо. При грамотном подходе к проектированию самое главное — выяснить, какой процесс кажется вам и вашей организации удобным, и придерживаться его. Нет смысла внедрять процесс проектирования, который никто не будет соблюдать.

Большинство книг, в которых рассматриваются объектно-ориентированные технологии, предлагает очень схожие стратегии проектирования систем. Фактически, за исключением некоторых из затрагиваемых специфических объектно-ориентированных вопросов, многое из стратегий также применимо к не объектно-ориентированным системам.

Как правило, надлежащий процесс объектно-ориентированного проектирования включает следующие этапы.

1. Проведение соответствующего анализа.
2. Составление технического задания, описывающего систему.
3. Сбор требований, исходя из составленного технического задания.
4. Разработка прототипа интерфейса пользователя.
5. Определение классов.
6. Определение ответственности каждого класса.
7. Определение того, как разные классы будут взаимодействовать друг с другом.
8. Создание высокоуровневой модели, описывающей систему, которую требуется построить.

В сфере объектно-ориентированной разработки высокоуровневая модель представляет особый интерес. Систему или объектную модель образуют диаграммы и взаимодействия классов. Эта модель должна точно представлять систему и быть легкой для понимания и модификации. Кроме того, необходима нотация для модели. Именно здесь в дело вступает унифицированный язык моделирования — Unified Modeling Language (UML). Как вы уже знаете, UML — это не процесс проектирования, а средство моделирования. В данной книге я сосредотачиваюсь только на диаграммах классов в рамках UML. Мне нравится использовать диаграммы классов в качестве визуального средства, которое помогает при проектировании и документировании.

ПОСТОЯННЫЙ ПРОЦЕСС ПРОЕКТИРОВАНИЯ

Несмотря на тщательное планирование, почти во всех случаях проектирование оказывается постоянным процессом. Даже после того как продукт будет протестирован, неожиданно возникнет необходимость внести конструктивные изменения. Менеджер проекта должен решить, где провести границу, которая укажет, когда следует перестать вносить изменения в продукт и добавлять функции.

Важно осознавать, что сейчас доступно множество методологий проектирования. Одна из первых методологий, называемая *моделью водопада*, предполагает установление точных границ между разными стадиями. При этом стадия проектирования должна завершиться до стадии реализации, которая, в свою очередь,

должна быть завершена до стадии тестирования и т. д. На практике модель водопада была признана нереалистичной. В настоящее время другие модели проектирования, например быстрое прототипирование, экстремальное программирование, гибкая разработка, Scrum и пр., пропагандируют истинный итеративный процесс. В этих моделях в качестве своего рода эксперимента предпринимается попытка пройти часть стадии реализации еще до завершения стадии проектирования. Несмотря на нынешнюю антипатию к модели водопада, цель этой модели понятна. Полностью и тщательно спроектировать все до начала написания кода — это рациональный подход. Наверняка вы не захотите снова пройти стадию проектирования, находясь на стадии выпуска продукта. Итерации с пересечением границ между стадиями неизбежны, однако вам следует сводить их к минимуму (рис. 6.1).

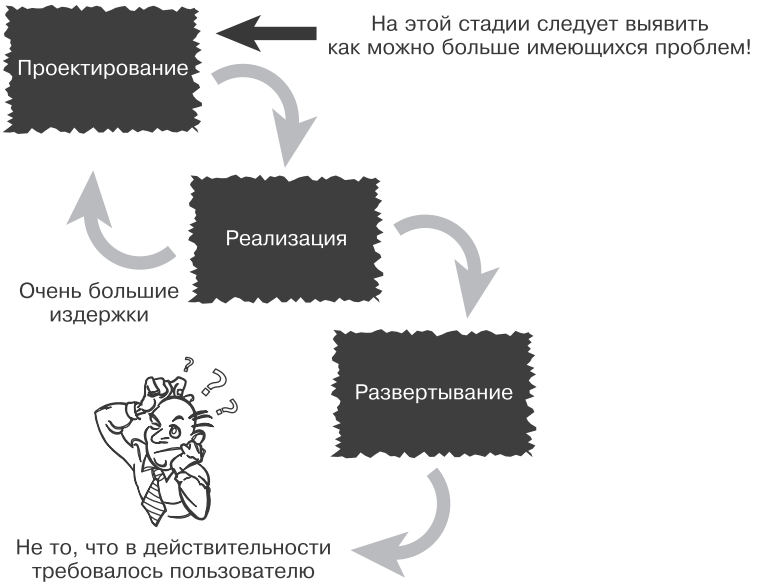

Рис. 6.1. Метод водопада

Попросту говоря, можно назвать следующие причины для заблаговременного определения требований и сведения конструктивных изменений к минимуму.

- Издержки из-за изменений требований/конструктивных правок на стадии проектирования будут сравнительно невелики.
- Издержки из-за конструктивных изменений на стадии реализации будут значительно выше.
- Издержки из-за конструктивных изменений после завершения стадии развертывания будут астрономическими по сравнению с теми, что упоминались в первом пункте.

Аналогичным образом вы не захотите начинать строительство дома своей мечты до того, как будет завершено архитектурное проектирование. Если я заявлю,

что мост Золотые Ворота или небоскреб Эмпайр-стейт-билдинг были построены без предварительного решения каких-либо задач проектирования, то вы подумаете, что это высказывание абсолютно безумно. Вместе с тем вы, скорее всего, не посчитаете мои слова глупыми, если я скажу вам, что используемое вами программное обеспечение может содержать конструктивные недостатки и, возможно, на самом деле не было тщательно протестировано.

В действительности может получиться так, что невозможно тщательно протестировать программное обеспечение, чтобы выявить *абсолютно все* дефекты. Однако в теории к этому следует стремиться. Вы должны всегда стараться устранить как можно больше имеющихся дефектов. Мосты и программное обеспечение, возможно, нельзя сравнивать напрямую, однако при работе с программным обеспечением нужно стремиться к тому же уровню конструкторского совершенства, что и в более «сложных» инженерных отраслях вроде строительства мостов. Использование программного обеспечения низкого качества может привести к фатальным последствиям — здесь имеются в виду не просто неправильные цифры на чеках по расчету заработной платы. Например, плохое программное обеспечение, заложенное в медицинское оборудование, может убить или покалечить людей. Кроме того, вы, возможно, будете готовы смириться с необходимостью время от времени перезагружать свой компьютер. Однако нельзя сказать то же самое, если речь идет об угрозе обрушения моста.

БЕЗОПАСНОСТЬ В ПРОТИВОПОСТАВЛЕНИИ С ЭКОНОМИЕЙ

Вы хотели бы перейти через мост, который не был тщательно испытан? К сожалению, во многих программных пакетах на пользователей возлагается обязанность по выполнению значительной части тестирования. Это обходится очень дорого как пользователям, так и поставщикам программного обеспечения. К сожалению, похоже, что краткосрочная экономия зачастую оказывается главным фактором при принятии решений о проектах.

Поскольку клиенты, по-видимому, согласны платить ограниченную цену и мириться с программным обеспечением низкого качества, некоторые поставщики ПО считают, что в долгосрочной перспективе будет дешевле позволять заказчикам тестировать продукт, нежели самим заниматься этим. Это, возможно, и верно, если говорить о ближайшей перспективе, однако в долгосрочной перспективе это обойдется намного дороже. В конечном счете пострадает репутация самих поставщиков.

После того как программное обеспечение будет выпущено, решение проблем, которые не были выявлены и устранены до выпуска продукта, обойдется намного дороже. В качестве наглядного примера возьмем дилемму, перед которой стоят автомобильные компании, столкнувшиеся с необходимостью отозвать из продажи свою продукцию. Если дефект в автомобилях будет выявлен и устранен до того, как они поступят в продажу, то это обойдется значительно дешевле, чем если все доставленные автомобили придется отзывать и ремонтировать поодиночке. Этот сценарий не только дорого обойдется, но и нанесет урон репутации компании. На рынке с постоянно растущей конкуренцией важно выпускать высококачественное программное обеспечение (рис. 6.2).

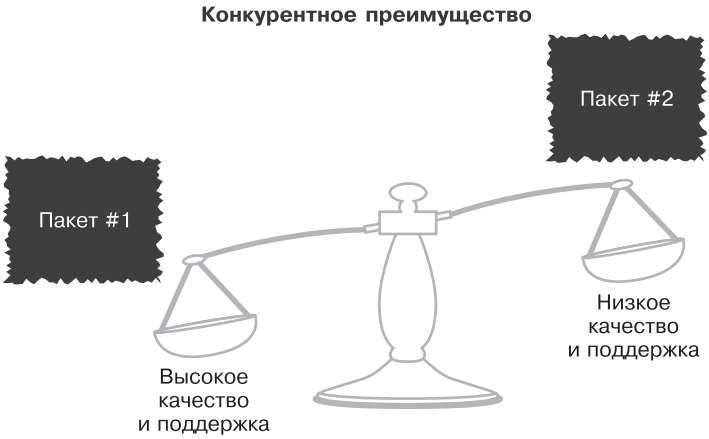

Рис. 6.2. Конкурентное преимущество

В последующих разделах кратко рассматриваются этапы процесса проектирования. Позднее в этой главе мы взглянем на пример, который подробнее объясняет каждый из этих этапов.

Проведение соответствующего анализа

В проектирование и производство того или иного программного продукта вовлечено много переменных факторов. Пользователи должны действовать рука об руку с разработчиками на всех стадиях. На стадии анализа пользователям и разработчикам необходимо провести соответствующее исследование и анализ, чтобы определить техническое задание, требования к проекту и понять, следует ли вообще заниматься этим проектом. Последний пункт может показаться немного неожиданным, однако он важен. На стадии анализа нужно без всяких колебаний прекратить работу над проектом, если выяснится, что на то есть веская причина. Слишком часто бывает так, что статус любимого проекта или некая политическая инертность способствуют поддержанию жизни в проекте вопреки очевидным предупреждающим знакам, которые «кричат» о необходимости его отмены. Если проект жизнеспособен, то основное внимание всех его участников на стадии анализа должно быть сосредоточено на изучении систем (как старой, так и предлагаемой новой), а также на определении системных требований.

ОБЩИЕ ПРИНЦИПЫ РАЗРАБОТКИ ПРОГРАММНОГО ОБЕСПЕЧЕНИЯ

Большинство этих методик нехарактерны для объектно-ориентированной разработки. Они относятся к разработке программного обеспечения в целом.

Составление технического задания

Техническое задание — документ, описывающий систему. Хотя определение требований — это конечная цель стадии анализа, на данном этапе они еще не обретают свою финальную форму. Техническое задание должно обеспечить полное понимание системы для любого человека, прочитавшего этот документ. Независимо

от того, как оно будет составлено, техническое задание должно представлять полную систему и ясно описывать то, как система будет выглядеть.

Техническое задание содержит всю информацию, что следует знать о системе. Многие заказчики готовят *заявку на проект* для распространения, которая схожа с техническим заданием. Заказчик формирует заявку на проект, полностью описывающую систему, создание которой ему необходимо, и распространяет эту заявку среди большого количества поставщиков. Поставщики затем используют этот документ при любом анализе, который им потребуется провести, чтобы выяснить, следует ли им браться за этот проект, и если да, то какую цену назначить за его выполнение.

Сбор требований

Документ с требованиями описывает, что, по мнению пользователей, должна делать система. Не обязательно излагать требования на высоком техническом уровне, но они должны быть достаточно конкретными для того, чтобы можно было понимать потребности пользователя в конечном продукте. Документ с требованиями должен быть достаточно подробным, чтобы пользователь затем смог вынести обоснованное решение о полноте системы.

В то время как техническое задание пишется с разделением на абзацы (или даже в повествовательной форме), требования обычно представляются в виде краткой сводки либо маркированного списка. Каждый пункт списка представляет одно определенное требование к системе. Требования «извлекаются» из технического задания. Этот процесс будет продемонстрирован позднее в текущей главе.

Во многих отношениях эти требования являются наиболее важной частью системы. Техническое задание может содержать не относящуюся к делу информацию, а требования — это итоговое представление системы, которое должно быть претворено в жизнь. Все будущие документы в процессе разработки программного обеспечения будут базироваться на этих требованиях.

Разработка прототипа интерфейса пользователя

Один из наилучших способов убедиться в том, что пользователям и разработчикам понятна система, — создать *прототип*. Прототип может быть практически всем чем угодно, однако большинство людей рассматривают его как имитацию интерфейса пользователя. Увидев фактические экраны и их последовательности, люди быстрее поймут, с чем им предстоит работать и как будет выглядеть система. Так или иначе, прототип почти наверняка не будет содержать всю функциональность итоговой системы.

Большинство прототипов создается с помощью той или иной *интегрированной среды разработки*. Visual Basic .NET традиционно является хорошей средой для прототипирования, хотя в настоящее время для этого также используются другие языки программирования. Помните, что вам не обязательно создавать бизнес-логику (логику, которая лежит в основе интерфейса и в действительности выполняет всю работу) при построении прототипа, хотя это возможно. На данном этапе главная забота — внешний вид интерфейса пользователя. Наличие прототипа может очень помочь при определении классов.

Определение классов

После того как требования будут задокументированы, вы сможете начать процесс определения классов. Если брать за основу требования, то самый простой способ определить классы — выделить все существительные. Они представляют людей, места и вещи. Не слишком беспокойтесь насчет того, чтобы определить сразу все классы. Может получиться так, что вам придется удалять, добавлять и изменять классы на разных стадиях всего процесса проектирования. Важно сначала что-нибудь написать. Помните, что проектирование является итеративным процессом. Как и в случае с другими формами мозгового штурма, писать изначально следует с осознанием того, что итоговый результат может быть абсолютно не похож на то, как все представлялось в самом начале.

Определение ответственности каждого класса

Вам потребуется определить ответственность каждого созданного ранее класса. Сюда входят данные, которые должен содержать класс, а также операции, которые он должен выполнять. Например, объект `Employee` отвечает за расчет заработной платы и перевод денег на соответствующий счет. Он также может отвечать за хранение таких данных, как разные уровни оплаты труда и номера счетов в различных банках.

Определение взаимодействия классов друг с другом

Большинство классов не существуют в изоляции. Хотя класс должен нести определенную ответственность, ему неоднократно придется взаимодействовать с другими классами, чтобы получить требуемое. Именно здесь находят свое применение сообщения между классами. Один класс может отправить сообщение другому, когда ему нужна информация из этого класса либо требуется, чтобы другой класс что-то сделал для него.

Создание модели классов для описания системы

Когда все классы, их ответственность и взаимодействия будут определены, вы сможете приступить к конструированию модели классов, представляющей полную систему. Модель классов показывает, как разные классы взаимодействуют в рамках системы.

В этой книге для моделирования системы мы используем UML. На рынке можно найти несколько инструментов, которые задействуют UML и обеспечивают хорошую среду для создания и сопровождения моделей классов UML. По мере рассмотрения примера в следующем разделе мы увидим, как диаграммы классов вписываются в общую картину и почему моделирование больших систем будет фактически невозможным без хорошей нотации.

Прототипирование интерфейса пользователя

Во время проектирования нам потребуется создать прототип нашего интерфейса пользователя. Этот прототип будет предоставлять бесценную информацию,

которая поможет в навигации по итерациям процесса проектирования. Как удачно подметили Гилберт и Маккарти в своей книге «Объектно-ориентированное проектирование на Java», «для пользователя системы пользовательский интерфейс является системой». Есть несколько способов создания прототипа интерфейса пользователя. Вы можете сделать набросок интерфейса на бумаге или лекционной доске либо воспользоваться специальным инструментом прототипирования или даже языковой средой вроде Visual Basic, которая часто применяется для быстрого прототипирования. Кроме того, вы можете прибегнуть к интегрированной среде разработки из вашего любимого средства разработки для создания прототипа.

Однако при разработке прототипа интерфейса пользователя позаботьтесь о том, чтобы у пользователей было право окончательного решения по поводу его внешнего вида.

Объектные обертки

В предыдущих главах я несколько раз отмечал, что одна из моих основных целей в этой книге — развеять миф, согласно которому объектно-ориентированное программирование является парадигмой, отдельной от структурного программирования, и даже противоречит ему. Более того, как я уже отмечал ранее, мне часто задают следующий вопрос: «Вы занимаетесь объектно-ориентированным или структурным программированием?» Ответ всегда звучит одинаково: «Я занимаюсь и тем и другим!»

Я считаю, что писать программы без использования структурированного кода невозможно. Таким образом, когда вы пишете программу на объектно-ориентированном языке и используете соответствующие методики объектно-ориентированного проектирования, вы также пользуетесь методиками структурного программирования. Это неизбежно.

Например, когда вы создадите новый объект, содержащий атрибуты и методы, эти методы будут включать структурированный код. Фактически я даже могу сказать, что эти методы будут содержать *в основном* структурированный код. Этот подход соответствует контейнерной концепции, с которой мы сталкивались в предыдущих главах. Кроме того, когда я подхожу к точке, в которой пишу код на уровне методов, мое мышление программиста незначительно меняется по сравнению с тем моментом, когда я программировал на структурных языках, например COBOL, C и т. п. Я не хочу сказать, что оно остается точно таким же, поскольку мне, несомненно, приходится привыкать к кое-каким объектно-ориентированным концепциям. Но основной подход к написанию кода на уровне методов практически не меняется.

Теперь я вернусь к вопросу: «Вы занимаетесь объектно-ориентированным или структурным программированием?» Я люблю говорить, что *программирование — это программирование*. Под этим я подразумеваю, что хороший программист понимает основы логики программирования и испытывает страсть к написанию кода.

Вы часто будете встречать объявления о найме программистов, обладающих набором определенных навыков, скажем знающих определенный язык программирования, например Java. Я четко осознаю, что той или иной организации в трудную минуту вполне может потребоваться опытный Java-программист, но если говорить о долгосрочной перспективе, то я бы предпочел сосредоточить внимание на найме программиста, имеющего большой опыт в области программирования и способного быстро учиться и приспосабливаться к новым технологиям. Некоторые из моих коллег не всегда соглашаются с этим; тем не менее я считаю, что при найме следует больше смотреть на то, чему потенциальный работник способен научиться, нежели на то, что он уже знает. Такая составляющая, как страсть к написанию кода, крайне важна, ведь она гарантирует, что работник будет постоянно исследовать новые технологии и методологии разработки.

Структурированный код

Несмотря на то что по поводу основ логики программирования возможны дебаты, как я уже подчеркивал, фундаментальными объектно-ориентированными концепциями являются *инкапсуляции*, *наследование*, *полиморфизм* и *композиция*. В большинстве учебников, которые я видел, в качестве базовых концепций структурного программирования указываются *последовательность*, *условия* и *итерации*.

Последовательность является базовой концепцией потому, что представляется логичным начинать сверху и следовать вниз. Для меня суть структурного программирования заключается в условиях и итерациях, которые я называю соответственно операторами `if` и циклами.

Взгляните на приведенный далее Java-код, который начинает с `0` и выполняет цикл десять раз, выводя значение, если оно равняется `5`:

```
class MainApplication {

  public static void main(String args[]) {

    int x = 0;

    while (x <= 10) {

      if (x==5) System.out.println("x = " + x);

      x++;
    }

  }

}
```

Несмотря на то что это объектно-ориентированный язык, код, располагающийся внутри основного метода, является структурированным. Присутствуют все три базовые концепции структурного программирования: *последовательность*, *условия* и *итерации*.

Такую составляющую, как последовательность, легко идентифицировать, поскольку первой выполняется строка:

```
int x = 0;
```

Когда выполнение этой строки завершается, выполняется следующая строка:

```
while (x <= 10) {
```

И так далее. Одним словом, это проверенный подход в виде нисходящего программирования: начать с первой строки, выполнить ее, а затем перейти к следующей.

В этом коде также содержится условие как часть оператора `if`:

```
if (x==5)
```

И наконец, цикл дополняет структурированное трио:

```
while (x <= 10) {
}
```

Вообще-то цикл `while` тоже содержит условие:

```
(x <= 10)
```

Вы можете написать довольно большое количество кода, руководствуясь лишь тремя этими концепциями. Фактически понятие обертки в структурном программировании в основном такое же, что и в объектно-ориентированном. При структурном проектировании в качестве оберток для кода используются функции (как, например, основной метод в рассмотренном примере), а при объектно-ориентированном проектировании обертками для кода выступают объекты и методы.

Обертывание структурированного кода

Хотя определение атрибутов считается написанием кода (например, создание целочисленной переменной), поведения объектов располагаются внутри методов. И в этих методах сосредоточена основная логика кода.

Взгляните на рис. 6.3. Как вы можете видеть, объект содержит методы, а они, в свою очередь, включают код, который может быть любым, начиная с объявлений переменных и заканчивая условиями и циклами.

Рассмотрим простой пример, в котором мы осуществим обертывание функциональности, обеспечивающей сложение. В данном случае мы создадим метод с именем `add`, который примет два целочисленных параметра и возвратит их сумму:

```
class SomeMath {

  public int add(int a, int b) {

    return a + b;

  }

}
```

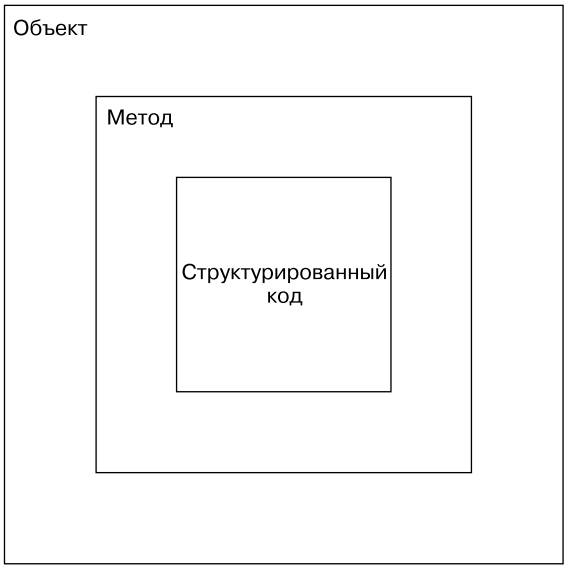

Рис. 6.3. Обертывание структурированного кода

Как вы можете видеть, структурированный код, используемый для выполнения сложения (`a+b`), *обернут* в метод `add`. Хотя это простой пример, в нем показано все, что касается обертывания структурированного кода. Таким образом, когда пользователь захочет использовать данный метод, ему потребуется лишь подпись этого метода, как показано далее:

```
public class TestMath {

  public static void main(String[] args) {

    int x = 0;

    SomeMath math = new SomeMath();

    x = math.add(1,2);

    System.out.println("x = " + x);

  }
}
```

И наконец, взглянем на еще одну функциональность, которая немного интереснее и сложнее. Допустим, нам понадобилось включить метод для вычисления значения числа Фибоначчи. Тогда мы можем добавить такой метод:

```
public static int fib(int n) {
  if (n < 2) {
    return n;
```

```
    } else {
      return fib(n-1)+fib(n-2);
    }
  }
```

Самое главное здесь — показать, что у нас имеется объектно-ориентированный метод, который содержит (обертывает собой) структурированный код, поскольку метод `fib` содержит условия, рекурсию и т. д. И, как уже отмечалось, в обертки также можно заключать существующий унаследованный код.

Обертывание непереносимого кода

Объектные обертки также могут использоваться для скрытия непереносимого (или нативного) кода. Концепция, в принципе, будет аналогичной, однако в данном случае суть заключается в том, чтобы взять код, выполнение которого возможно только на одной платформе (или немногих платформах), и инкапсулировать его в методе с обеспечением простого интерфейса для программистов, которые будут использовать этот код.

Возьмем, к примеру, задачу подачи *сигнала*. На платформе Windows мы можем обеспечить подачу *сигнала* с помощью приведенного далее кода:

```
System.out.println("\007");
```

Вместо того чтобы вынуждать программистов запоминать код (или искать его), вы можете предусмотреть класс с именем `Sound`, содержащий метод `beep`, как показано далее:

```
class Sound {

  public void beep() {

    System.out.println("\007");

  }

}
```

Теперь вместо того, чтобы запоминать код для обеспечения подачи сигнала, программисты смогут использовать этот класс и вызывать метод `beep`:

```
public class TestBeep {

  public static void main(String[] args) {

    Sound mySound = new Sound();

    mySound.beep();

  }
}
```

Так программистам будет проще работать. Кроме того, вы сможете расширить функциональность класса, включив в него другие методы для генерирования

сигналов. Пожалуй, более важно то, что при работе кода на платформе, не являющейся Windows, интерфейс для пользователей останется прежним. Коротко говоря, команде, которая будет создавать код для класса `Sound`, придется иметь дело с изменением платформы. Для программистов, использующих этот класс в своих приложениях, изменение не создаст никаких проблем, поскольку они по-прежнему смогут вызывать метод `beep`.

Обертывание существующих классов

Хотя необходимость обертывать унаследованный структурированный или даже непереносимый код в тот или иной новый (объектно-ориентированный) класс может показаться резонной, необходимость обертывать существующие классы уже не настолько очевидна. Кроме того, есть много причин, чтобы создавать обертки для существующих (объектно-ориентированных) классов.

Разработчики программного обеспечения часто применяют код, написанный кем-то другим. Это может быть код, приобретенный у поставщика или даже написанный людьми из той же организации. Во многих случаях оказывается, что код нельзя изменить. Возможно, из-за того, что человек, написавший код, больше не работает в организации либо поставщик не может предоставить пакеты обновлений и т. д. Именно в таких ситуациях проявляется истинная мощь оберток.

Идея заключается в том, чтобы взять существующий класс и изменить его реализацию или интерфейс, обернув его в новый класс, — точно так же, как мы делали это со структурированным и непереносимым кодом. Разница здесь состоит в том, что вместо того, чтобы придать коду объектно-ориентированное «обличье», мы изменяем его реализацию или интерфейс.

Зачем нам это может потребоваться? Что ж, ответ заключается как в реализации, так и в интерфейсе.

Вспомните пример с базой данных, который мы использовали в главе 2. Наша цель состояла в обеспечении одинакового интерфейса для разработчиков независимо от того, какую базу данных они будут использовать. Фактически, если бы нам потребовалось обеспечить поддержку другой базы данных, наша цель осталась бы прежней — сделать переход на новую базу данных прозрачным для пользователей (см. рис. 2.3).

Кроме того, вспомните, как мы рассматривали вопрос создания промежуточного программного обеспечения для предоставления интерфейса между объектами и реляционными базами данных. Нам как разработчикам требуется использовать объекты. Таким образом, понадобится функциональность, которая позволит нам сохранять объекты в базе данных. Мы не хотим, чтобы нам пришлось писать SQL-код для каждой объектной транзакции, осуществляемой при работе с реляционной базой данных. Вот где мы можем считать промежуточное программное обеспечение оберткой. При этом доступно много продуктов для объектно-реляционного отображения. Эта тема намного подробнее рассматривается в главе 12.

В принципе, для меня идеальным примером парадигмы «интерфейс/реализация» является исследование, которое мы проводили в примере с электростанцией в главе 2 (см. рис. 2.1). В данном случае у нас есть возможность изменить (обернуть) и то и другое: мы можем изменить интерфейс, сменив электрическую розетку, и можем изменить реализацию, сменив электростанцию.

Обертки довольно широко используются при разработке программного обеспечения, причем с позиции не только разработчиков, но и поставщиков. Обертки — это важный инструмент при создании программных систем.

Резюме

В этой главе был рассмотрен процесс проектирования полных систем. Важно отметить, что объектно-ориентированный и структурированный код не являются взаимоисключающими. Более того, вы не сможете создавать объекты без использования структурированного кода. Таким образом, при создании объектно-ориентированных систем в процессе проектирования вы также будете использовать структурные методики.

Объектные обертки применяются для инкапсуляции функциональности многих типов, которая может варьироваться от традиционного структурированного (унаследованного) и объектно-ориентированного (классы) кода до непереносимого (нативного) кода. Основное назначение объектных оберток — обеспечивать согласованные интерфейсы для программистов, использующих соответствующий код.

В последующих главах мы подробнее исследуем взаимоотношения между классами. В главе 7 рассматриваются концепции наследования и композиции, а также то, как они соотносятся друг с другом.

Ссылки

- *Амблер Скотт*. Введение в объектно-ориентированную технологию (The Object Primer). 3-е изд. — Кембридж: Cambridge University Press, 2004.
- *Макконнелл Стив*. Совершенный код: практическое руководство по разработке программного обеспечения (Code Complete: A Practical Handbook of Software Construction). 2-е изд. — Редмонд: Microsoft Press, 2004.
- *Гилберт Стивен и Маккарти Билл*. Объектно-ориентированное проектирование на Java (Object-Oriented Design in Java). — Беркли: The Waite Group Press (Pearson Education), 1998.
- *Яворски Джейми*. Платформа Java 2 в действии (Java 2 Platform Unleashed). — Индианаполис: Sams Publishing, 1999.
- *Яворски Джейми*. Руководство разработчика на Java 1.1 (Java 1.1 Developers Guide). — Индианаполис: Sams Publishing, 1997.
- *Вирфс-Брок Р., Вилкерсон Б. и Вейнер Л.* Проектирование объектно-ориентированного программного обеспечения (Designing Object-Oriented Software). — Аппер-Сэддл-Ривер: Prentice-Hall, 1997.
- *Вайсфельд Мэтт и Киккоцци Джон*. Разработка программного обеспечения рабочей группой (Software by Committee). / Project Management, — Сентябрь 1999. — Том 5, № 1. — С. 30–36.

Глава 7
Наследование и композиция

Наследование и композиция играют главные роли в проектировании объектно-ориентированных систем. Фактически многие из наиболее сложных и интересных проектных решений сводятся к выбору между наследованием и композицией.

С годами эти решения становились намного интереснее по мере того, как развивались методики объектно-ориентированного проектирования. Пожалуй, наиболее любопытные дебаты ведутся вокруг наследования. Хотя наследование — это одна из фундаментальных концепций объектно-ориентированной разработки (язык программирования должен поддерживать наследование для того, чтобы считаться объектно-ориентированным), многие программисты все чаще и чаще отказываются от наследования в пользу других стратегий проектирования.

Несмотря на это как наследование, так и композиция представляют собой механизмы повторного использования. *Наследование*, как видно из его названия, подразумевает получение по наследству атрибутов и поведений от других классов. При этом имеет место настоящее отношение «родительский класс/дочерний класс». Дочерний класс (или подкласс) наследует напрямую от родительского класса (или суперкласса).

Композиция, как тоже видно из названия, подразумевает создание объектов с использованием других объектов. В этой главе мы исследуем явные и тонкие различия между наследованием и композицией. В первую очередь мы рассмотрим, когда именно и какой механизм следует использовать.

Повторное использование объектов

Пожалуй, главная причина существования наследования и композиции — повторное использование объектов. Коротко говоря, вы можете создавать классы (которые в конечном счете станут объектами), применяя другие классы посредством наследования и композиции. Ведь эти механизмы фактически являются единственными способами повторного использования ранее созданных классов.

Наследование представляет отношение «является экземпляром», рассмотренное в главе 1. Например, собака *является экземпляром* млекопитающего.

Композиция подразумевает использование других классов для создания более сложных классов, то есть для осуществления своего рода сборки. При этом нет никаких отношений «родительский класс/дочерний класс». По сути сложные объекты состоят из других объектов. Композиция представляет отношение «содержит как часть». Например, автомобиль *содержит как часть* двигатель. Двигатель и автомобиль — отдельные, потенциально самостоятельные объекты. Однако

автомобиль является сложным объектом, который включает в себя (содержит как часть) такой объект, как двигатель. Кроме того, дочерний объект сам может состоять из других объектов; например, двигатель может включать в себя цилиндры. При этом двигатель *содержит как часть* цилиндр, которых на самом деле несколько.

Когда объектно-ориентированные технологии только начали получать широкое распространение, наследование часто оказывалось первым, к чему программисты прибегали при проектировании объектно-ориентированных систем. Возможность единожды спроектировать класс, а затем наследовать от него функциональность считалась одним из главных преимуществ использования объектно-ориентированных технологий.

Однако в дальнейшем слава наследования немного померкла. На самом деле даже во время первых дискуссий использование наследования как такового ставилось под сомнение. В книге «Проектирование на Java» *(Java Design)* под авторством Петера Коуда и Марка Мейфилда имеется целая глава под названием «Проектирование с использованием композиции вместо наследования». Многие ранние объектно-ориентированные платформы даже не поддерживали истинного наследования. Когда Visual Basic превращался в Visual Basic .NET, первые объектно-ориентированные реализации не содержали возможностей по строгому наследованию. Платформы вроде модели COM от компании Microsoft базировались на наследовании интерфейсов (оно подробно рассматривается в главе 8).

В настоящее время использование наследования по-прежнему остается важной темой дебатов. Абстрактные классы, представляющие собой форму наследования, не поддерживаются напрямую в некоторых языках программирования, например Objective-C. Интерфейсы используются, даже если они не обеспечивают всей функциональности, которую предоставляют абстрактные классы.

Хорошая новость заключается в том, что дискуссии насчет того, применять наследование либо композицию, полезны, ведь они направлены на выработку взвешенного компромиссного решения. Как и во всех философских дебатах, обе стороны пылко приводят свои аргументы. К счастью, как это обычно бывает, эти горячие дискуссии привели к правильному пониманию того, как использовать соответствующие технологии.

Позднее в этой главе вы увидите, почему многие программисты считают, что наследования следует избегать и нужно предпочитать композицию. Аргументация довольно сложна и хитроумна. На самом деле как наследование, так и композиция являются действующими методиками проектирования классов, и у каждой из них есть соответствующее место в инструментарии разработчика, использующего объектно-ориентированные технологии. А вам как минимум необходимо понимать обе эти методики, чтобы сделать правильный выбор при проектировании.

Да, наследование часто неправильно используют или злоупотребляют им, но это происходит из-за непонимания того, что оно собой представляет. Нужно помнить, что и наследование, и композиция — важные методики при создании объектно-ориентированных систем. Проектировщикам и разработчикам необходимо потратить время на то, чтобы разобраться в преимуществах и недостатках обеих методик, и применять каждую из них в соответствующих ситуациях.

Наследование

В главе 1 наследование было определено как система, при которой дочерние классы наследуют атрибуты и поведения от родительского класса. Однако это еще не все, что можно сказать о наследовании, и в этой главе мы подробнее исследуем его.

Ранее отмечалось, что отношение наследования можно определить, придерживаясь простого правила: если вы можете сказать, что класс B *является экземпляром* класса A, то это отношение — хороший кандидат на то, чтобы быть отношением наследования.

ОТНОШЕНИЕ «ЯВЛЯЕТСЯ ЭКЗЕМПЛЯРОМ»

Одно из основных правил объектно-ориентированного проектирования заключается в том, что открытое наследование представляется отношением «является экземпляром».

Снова обратимся к примеру с классами млекопитающих, приводившемуся в главе 1. Взглянем на класс `Dog`. Он содержит несколько поведений, благодаря которым совершенно ясно, что `Dog` противоположен классу `Cat`. В этом примере укажем два поведения для `Dog`: `bark` и `pant`. Таким образом, мы сможем создать класс `Dog`, содержащий два поведения наряду с двумя атрибутами (рис. 7.1).

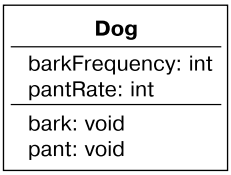

Рис. 7.1. Диаграмма класса Dog

Теперь предположим, что нам потребовалось создать класс `GoldenRetriever`. Можно создать совершенно новый класс, содержащий те же поведения, которые имеются в классе `Dog`. При этом можно будет сделать следующий вполне обоснованный вывод: `GoldenRetriever` является экземпляром `Dog`. В силу этого отношения у нас будет возможность наследовать атрибуты и поведения от `Dog` и использовать их в новом классе `GoldenRetriever` (рис. 7.2).

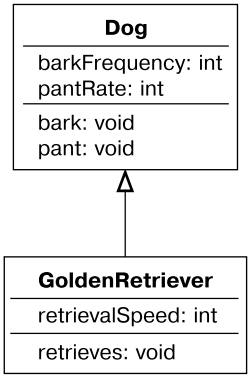

Рис. 7.2. Класс GoldenRetriever наследует от класса Dog

Класс GoldenRetriever теперь содержит собственные поведения, а также все более общие поведения класса Dog. Это обеспечивает для нас значительные преимущества. Во-первых, когда мы создавали класс GoldenRetriever, нам не пришлось делать часть того, что и так уже было сделано, то есть переписывать методы bark и pant. Это позволяет сэкономить время, которое уходит на проектирование и написание кода, а также на тестирование и сопровождение. Методы bark и pant пишутся только один раз, и при условии, что они были надлежащим образом протестированы при создании класса Dog, их не придется опять основательно тестировать. Вместе с тем их потребуется заново протестировать при добавлении новых интерфейсов и т. д.

Теперь полностью используем нашу структуру наследования и создадим второй класс, который будет дочерним по отношению к классу Dog. Назовем его LhasaApso. В то время как ретриверов разводят для того, чтобы использовать их в качестве охотничьих поисковых собак, тибетские терьеры предназначены для охраны. Это не боевые собаки; у них острый нюх, и когда эти собаки чуют что-то необычное, они начинают лаять. Таким образом, мы можем создать наш класс LhasaApso, который будет наследовать от класса Dog точно так же, как это делал класс GoldenRetriever (рис. 7.3).

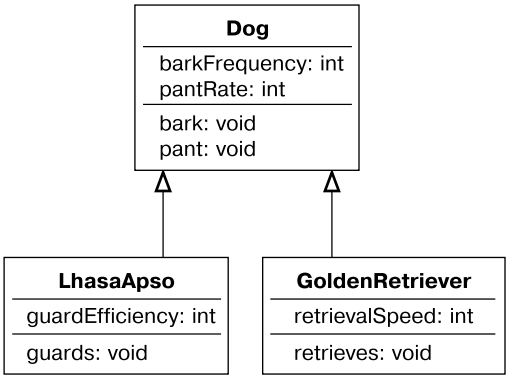

Рис. 7.3. Класс LhasaApso наследует от класса Dog

ТЕСТИРОВАНИЕ НОВОГО КОДА

В нашем примере с классом GoldenRetriever методы bark и pant должны быть написаны, протестированы и отлажены при создании класса Dog. Теоретически этот код теперь надежен и готов к повторному использованию в других ситуациях. Однако тот факт, что вам не нужно переписывать код, не означает, что вы не должны его протестировать. Конечно, маловероятно, что у GoldenRetriever имеется некая особенность, которая тем или иным образом нарушит код. Однако вам всегда следует тестировать новый код. Каждое новое отношение наследования создает контекст для использования унаследованных методов. Стратегия полного тестирования должна учитывать каждый из таких контекстов.

Еще одно основное преимущество наследования состоит в том, что код для bark() и pant() располагается в одном месте. Допустим, возникла необходимость изменить код в методе bark(). Когда вы измените его в классе Dog, вам не придется изменять его в классах LhasaApso и GoldenRetriever.

Вы видите здесь проблему? На этом уровне кажется, что модель наследования очень хорошо работает. Однако как вы можете быть уверены в том, что все собаки ведут себя именно так, как это отражает поведение, содержащееся в классе `Dog`?

В своей книге «Эффективное использование C++» *(Effective C++)* Скотт Майерс приводит отличный пример дилеммы, касающейся проектирования с использованием наследования. Рассмотрим класс с именем `Bird`. Одной из наиболее узнаваемых особенностей птиц является, конечно же, их способность летать. Таким образом, мы создадим класс с именем `Bird`, содержащий метод `fly`. Вам сразу же должна стать понятна проблема. Что нам делать с пингвином или страусом? Они тоже являются птицами, однако не умеют летать. Вы могли бы переопределить соответствующее поведение локально, однако метод по-прежнему будет иметь имя `fly`. Нет смысла располагать методом с именем `fly` для класса птиц, которые не летают, а только ходят вперевалку, бегают или плавают.

Это приводит к значительным проблемам. Например, в классе для пингвина содержится метод `fly`, который пингвину может захотеться опробовать по вполне понятным причинам. Однако если бы на самом деле метод `fly` оказался переопределен и такое поведение, как полет, отсутствовало бы, то пингвин очень удивился бы вызовом метода `fly`, прыгнув через крутой обрыв. Представьте себе разочарование пингвина, если вызов метода `fly` приведет к ходьбе вперевалку, а не к полету. В данной ситуации ходьба вперевалку не позволит справиться с поставленной задачей. Подумать только, что было бы, если бы такой код когда-нибудь оказался заложен в систему управления космического корабля.

В нашем примере с `Dog` мы спроектировали этот класс таким образом, что все дочерние по отношению к нему классы содержат метод `bark`, отражающий способность собак лаять. Однако некоторые собаки не лают. Например, собаки породы басенджи не умеют лаять. Несмотря на это они издают звуки, похожие на йодль. Как же нам следует пересмотреть то, что мы спроектировали? Как код выглядел бы после этого? На рис. 7.4 показан пример, демонстрирующий более корректный подход к моделированию иерархии класса `Dog`.

Обобщение и конкретизация

Взглянем на объектную модель иерархии классов во главе с `Dog`. Мы начали с одного класса с именем `Dog` и определили общность между разными породами собак. Эту концепцию, иногда называемую *обобщением-конкретизацией*, также необходимо принимать во внимание при использовании наследования. Идея заключается в том, что по мере того как вы спускаетесь по дереву наследования, все становится более конкретным. Самое общее располагается на верхушке дерева наследования. Если рассматривать наше дерево наследования `Dog`, класс с аналогичным названием располагается на его верхушке и является наиболее общей категорией. Разные породы — классы `GoldenRetriever`, `LhasaApso` и `Basenji` — являются наиболее конкретными. Идея наследования состоит в том, чтобы переходить от общего к частному, выделяя общность.

Работая с моделью наследования `Dog`, мы начали выделять общность в поведении, понимая при этом, что, хотя поведение ретривера отличается от поведения тибетского терьера, у этих пород есть кое-что общее — например, для тех и других собак

характерно учащенное дыхание. Кроме того, они лают. Затем мы осознали, что не все собаки лают — некоторые издают звуки, похожие на йодль. Таким образом, нам пришлось вынести поведение `bark` в отдельный класс `BarkingDog`. Поведение `yodels` мы поместили в класс `YodelingDog`. Однако мы осознали, что у всех лающих и нелающих собак все равно имеется кое-какая общность в поведении — все эти собаки учащенно дышат. Поэтому мы сохранили класс `Dog` и сделали так, чтобы от него наследовали классы `BarkingDog` и `YodelingDog`. Теперь `Basenji` может наследовать от `YodelingDog`, а у `LhasaApso` и `GoldenRetriever` есть возможность наследовать от `BarkingDog`.

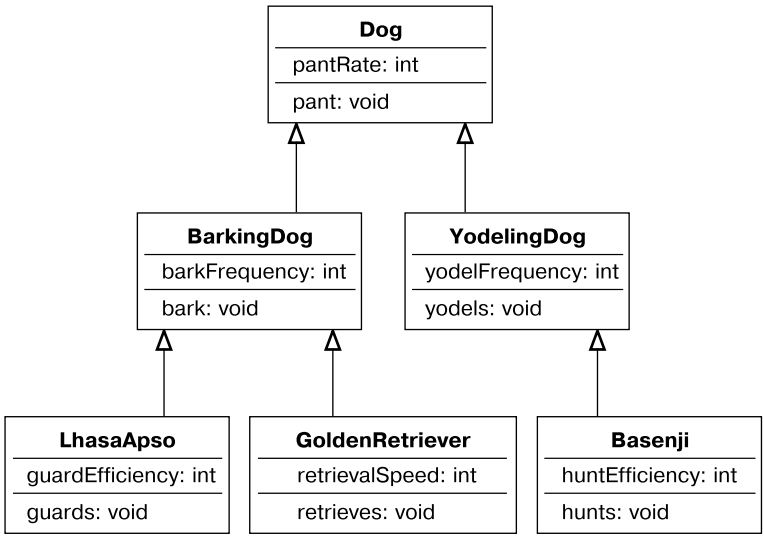

Рис. 7.4. Иерархия классов во главе с Dog

Мы могли бы решить не создавать два отдельных класса `BarkingDog` и `YodelingDog`. Тогда мы смогли бы реализовать `bark` и `yodels` как часть класса каждой отдельной породы, поскольку звуки, издаваемые собаками этих пород, различаются. Это лишь один пример проектных решений, которые придется принять. Пожалуй, наилучшим решением станет реализация `bark` и `yodels` как интерфейсов, о чем мы поговорим в главе 8.

Проектные решения

В теории лучший подход — выделение как можно большей общности. Однако, как и во всех задачах проектирования, иногда, как говорится, хорошего понемножку. Несмотря на то что выделение как можно большей общности может быть максимально приближенным к реальной жизни, оно может не быть максимально приближенным к вашей модели. Чем бо́льшую общность вы выделяете, тем сложнее становится ваша система. Таким образом, вам необходимо решить головоломку: вы хотите, чтобы у вас была более точная модель или же менее

сложная система? Вам придется сделать выбор в зависимости от вашей ситуации, и нет никаких жестких директив, которым необходимо следовать при принятии этого решения.

В ЧЕМ КОМПЬЮТЕРЫ НЕ СИЛЬНЫ

Ясно, что компьютерная модель способна лишь примерно отражать реальные ситуации. Компьютеры сильны в решении числовых задач большого объема, однако не так хороши в выполнении более абстрактных операций.

Например, разбиение класса `Dog` на `BarkingDog` и `YodelingDog` моделирует реальную жизнь лучше, чем допущение, что все собаки лают, однако такой подход все немного усложняет.

СЛОЖНОСТЬ МОДЕЛИ

Добавив еще два класса на данном уровне нашего примера, мы не усложним все настолько, что могло бы сделать модель неудачной. Однако при работе с более крупными системами, когда решения подобного рода принимаются много раз, сложность быстро повышается. Если говорить о крупных системах, то наилучшим подходом будет сохранение как можно большей простоты.

При проектировании вы будете сталкиваться с ситуациями, когда преимущества более точной модели не будут оправдывать повышение сложности. Предположим, что вы являетесь собаководом и заключили субподрядный договор на создание системы, позволяющей отслеживать всех ваших собак. Системная модель, которая включает собак как лающих, так и издающих звуки, похожие на йодль, отлично работает. Однако, допустим, вы не разводите собак, издающих звуки, похожие на йодль. Пожалуй, у вас не будет необходимости усложнять систему разграничением собак по указанному параметру. Это сделает систему более простой и обеспечит требуемую вам функциональность.

Решение о том, проектировать все так, чтобы система была менее сложной или же обладала большей функциональностью, должно быть сбалансированным. Основная цель заключается в том, чтобы всегда стремиться создать систему, которая будет гибкой, но не настолько сложной, что может рухнуть под собственной тяжестью.

Текущие и будущие издержки тоже относятся к числу основных факторов, влияющих на такие решения. Несмотря на то что желание сделать систему более полной и гибкой может показаться уместным, добавленная в результате этого функциональность едва ли привнесет какие-либо преимущества — окупаемости инвестиций может не произойти. Например, расширили ли бы вы конструкцию своей системы `Dog`, чтобы включить в нее других представителей семейства псовых вроде гиен и лис (рис. 7.5)?

Если вы являетесь служителем зоопарка, то такая конструкция может оказаться целесообразной, однако, если вы разводите и продаете домашних собак, расширять класс `Canine`, вероятно, не потребуется.

Как вы можете видеть, при проектировании всегда приходится принимать компромиссные решения.

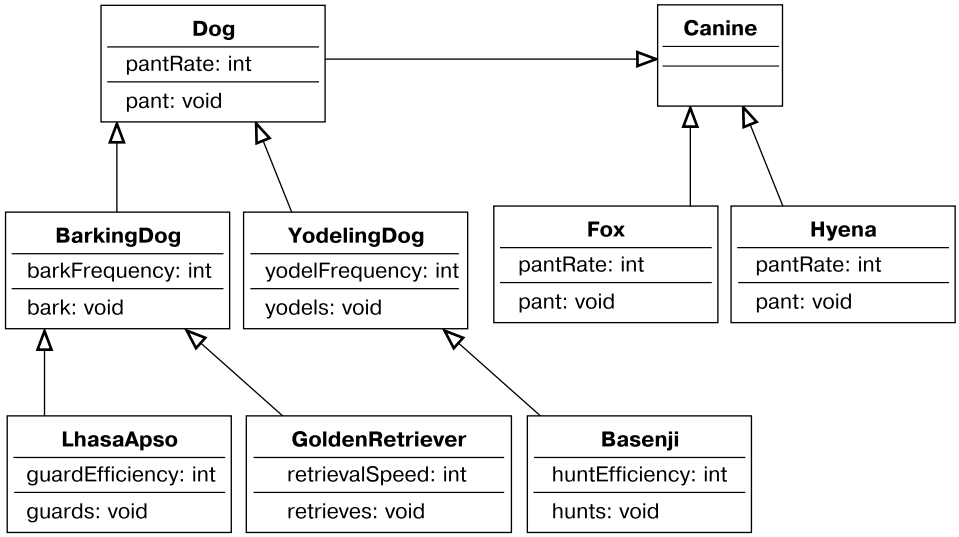

Рис. 7.5. Расширенная модель Canine

ПРИНЯТИЕ ПРОЕКТНЫХ РЕШЕНИЙ С УЧЕТОМ БУДУЩЕГО

На данном этапе вы могли бы сказать «Никогда не говори “никогда”». Несмотря на то что сейчас вы, возможно, не разводите собак, издающих звуки, которые напоминают йодль, когда-нибудь в будущем у вас может возникнуть желание заняться этим. Если вы не станете сразу проектировать систему с учетом возможности разведения таких собак, то в дальнейшем соответствующее изменение системы обойдется намного дороже. Это еще одно из многих проектных решений, которые вам придется принять. Вы, вероятно, могли бы переопределить метод bark() для того, чтобы «превратить» его в yodels(), однако результат не окажется интуитивно понятным, поскольку некоторые люди ожидают, что метод, позволяющий выполнять соответствующее действие, будет иметь имя bark().

Композиция

Вполне естественно представлять себе, что в одних объектах содержатся другие объекты. У телевизора есть тюнер и экран. У компьютера есть видеокарта, клавиатура и накопитель. Компьютер можно считать объектом, но и флэш-диск тоже считается полноценным объектом. Вы могли бы открыть системный блок компьютера, снять жесткий диск и подержать его в руке. Более того, вы могли бы установить этот жесткий диск в другой компьютер. Утверждение, что этот накопитель является самостоятельным объектом, подкрепляется тем, что он может работать в разных компьютерах.

Классический пример композиции объектов — автомобиль. Похоже, что во многих книгах, статьях и на подготовительных курсах автомобиль используется как олицетворение композиции объектов. Большинство людей считают автомобильный сборочный конвейер, придуманный Генри Фордом, основным примером производства с использованием взаимозаменяемых деталей. Таким образом, кажется естественным, что автомобиль стал главным «исходным пунктом» при проектировании объектно-ориентированных программных систем.

Почти всем людям показалось бы вполне естественным, что автомобиль имеет двигатель. Однако в состав автомобиля также входит много других объектов, включая колеса, руль и стереосистему (рис. 7.6). Во всех случаях, когда определенный объект состоит из других объектов, которые включены как объектные поля, новый объект называется *составным*, *агрегированным* или *обобщенным*.

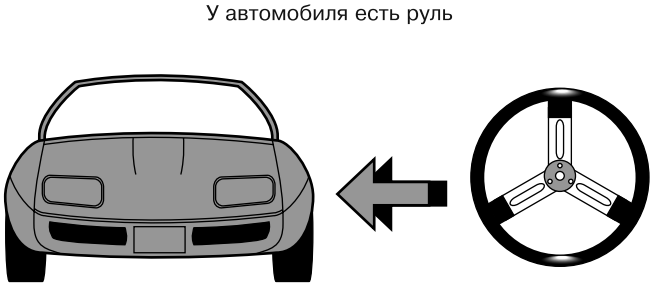

Рис. 7.6. Пример композиции

АГРЕГАЦИЯ, АССОЦИАЦИЯ И КОМПОЗИЦИЯ

На мой взгляд, есть только два способа повторного использования классов — с помощью наследования или композиции. В главе 9 мы подробно поговорим о композиции, в частности об агрегации и ассоциации. В этой книге я считаю агрегацию и ассоциацию типами композиции, хотя на этот счет есть разные мнения.

Представление композиции на UML. Моделируя на UML тот факт, что в состав объекта «автомобиль» входит объект «руль», задействуем нотацию, показанную на рис. 7.7.

АГРЕГАЦИЯ, АССОЦИАЦИЯ И UML

В этой книге агрегации представлены на UML линиями с ромбом, например для двигателя, являющегося частью автомобиля. Ассоциации обозначены просто линиями (без ромба), например для отдельной клавиатуры, обслуживающей отдельный системный блок компьютера.

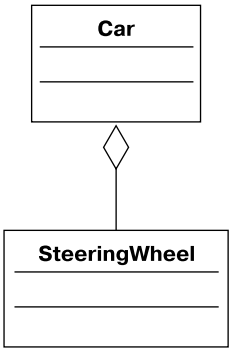

Рис. 7.7. Представление композиции на UML

Обратите внимание, что на конце линии, соединяющей класс `Car` с классом `SteeringWheel`, имеется ромб на стороне класса `Car`. Это означает, что `Car` *включает (содержит как часть)* `SteeringWheel`.

Расширим этот пример. Допустим, ни один из объектов в этой конструкции не задействует наследования. Все объектные отношения являются строго отношениями композиции, при этом есть ее разные уровни. Конечно, это упрощенный пример, а при проектировании автомобиля используется гораздо больше объектов и объектных отношений. Однако эта конструкция призвана послужить простой иллюстрацией того, что представляет собой композиция.

Скажем, автомобиль состоит из двигателя, стереосистемы и двери.

СКОЛЬКО ДВЕРЕЙ И СТЕРЕОСИСТЕМ?

Следует отметить, что у автомобиля обычно несколько дверей. У одних автомобилей их две, а у других — четыре. Вы даже можете посчитать пятой дверью ту, что расположена сзади у автомобиля с кузовом типа хетчбэк. В том же духе не в каждом автомобиле обязательно есть стереосистема. Мне даже доводилось видеть автомобили с двумя отдельными стереосистемами. Подобные ситуации подробно рассматриваются в главе 9.

Пока же для нашего примера договоримся, что у автомобиля есть только одна дверь (возможно, это особый гоночный автомобиль) и одна стереосистема.

То, что автомобиль состоит из двигателя, стереосистемы и двери, легко понять, поскольку большинство людей именно так и представляют себе автомобили. Однако при проектировании объектно-ориентированных программных систем важно помнить, что объекты, как и автомобили, состоят из других объектов. Более того, количество узлов и ветвей, которое может включать соответствующая древовидная структура классов, фактически неограниченно.

На рис. 7.8 показана объектная модель для `Car`, включая подклассы `Engine`, `Stereo` и `Door`.

Обратите внимание, что все три объекта, которые образуют `Car`, сами состоят из других объектов. `Engine` содержит `Pistons` и `SparkPlugs`; `Stereo` включает `Radio` и `Cassette`; `Door` содержит `Handle`. Заметьте также, что там имеется еще один уровень: `Radio` содержит `Tuner`. Мы могли бы также добавить, что `Handle` содержит `Lock`, а `Cassette` включает `FastForwardButton`. Кроме того, мы могли бы пойти на один уровень дальше `Tuner` и создать объект `Dial`.

Проектировщику решать, какими будут качество и сложность объектной модели.

СЛОЖНОСТЬ МОДЕЛИ

Как и в случае с проблемой наследования в примере с классами, которые касаются лающих и нелающих собак, злоупотребление композицией может привести к повышению сложности.

Между созданием объектной модели, достаточно детализированной для того, чтобы быть адекватно выразительной, и модели, которая настолько детализирована, что ее сложно понять и сопровождать, проходит тонкая грань.

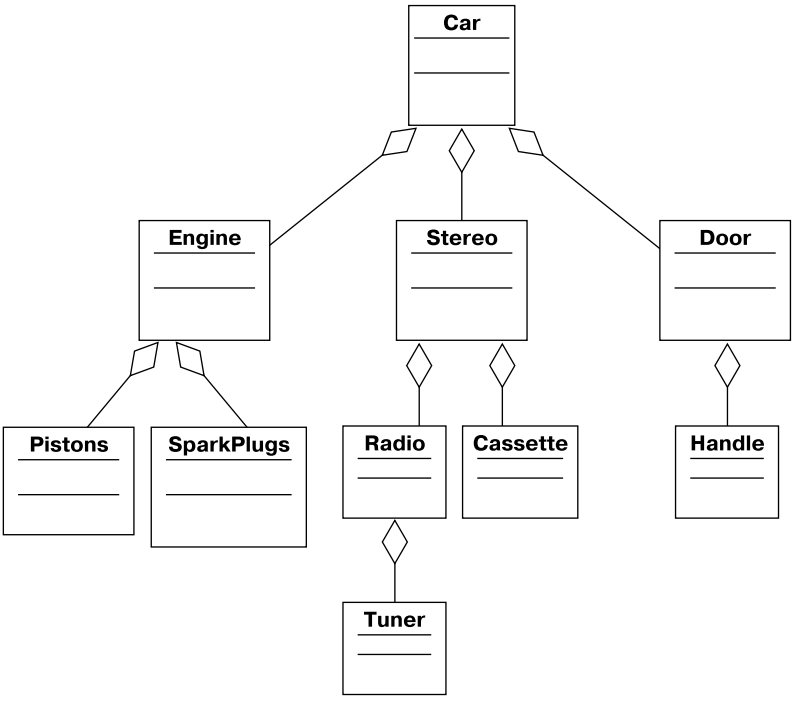

Рис. 7.8. Иерархия классов во главе с Car

Почему инкапсуляция является фундаментальной объектно-ориентированной концепцией

Инкапсуляция — фундаментальная объектно-ориентированная концепция. Каждый раз при рассмотрении парадигмы «интерфейс/реализация» мы говорим об инкапсуляции. Основной вопрос заключается в том, что в классе должно быть видно, а что — нет. Инкапсуляция в равной мере касается данных и поведений. Когда речь идет о классе, то первоочередное проектное решение «вращается» вокруг инкапсуляции как данных, так и поведений в хорошо написанном классе.

Гилберт и Маккарти определяют инкапсуляцию как «процесс упаковки вашей программы с разделением каждого из ее классов на две обособленные части — интерфейс и реализацию». Эта идея многократно повторяется и по ходу нашей книги.

Но при чем здесь инкапсуляция и какое отношение она имеет к этой главе? В данном случае мы имеем дело с объектно-ориентированным парадоксом. Инкапсуляция является настолько фундаментальной объектно-ориентированной концепцией, что представляет собой одно из главных правил ООП. Наследование тоже считается одной из трех важнейших объектно-ориентированных концепций. Однако оно некоторым образом фактически нарушает инкапсуляцию!

Как такое возможно? Неужели две из трех важнейших объектно-ориентированных концепций противоречат друг другу? Рассмотрим это подробнее.

Как наследование ослабляет инкапсуляцию

Как уже говорилось, инкапсуляция — это процесс упаковки классов в открытый интерфейс и закрытую реализацию. По сути в классе скрывается все, о чем другим классам знать не обязательно.

Петер Коуд и Марк Мейфилд отмечают, что при использовании наследования инкапсуляция, в сущности, ослабляется в рамках иерархии классов. Они говорят о конкретном риске: наследование означает сильную инкапсуляцию по отношению к остальным классам, но слабую инкапсуляцию между суперклассом и его подклассами.

Проблема заключается в том, что если от суперкласса будет унаследована реализация, которая затем подвергнется модификации, то такое изменение *распространится* по иерархии классов. Этот волновой эффект потенциально способен затронуть все подклассы. Поначалу это может не показаться большой проблемой, однако, как мы уже видели ранее, подобный волновой эффект может привести к непредвиденным проблемам. Например, тестирование превратится в кошмар. В главе 6 мы говорили о том, как инкапсуляция упрощает системы тестирования. В теории, если вы создадите класс с именем `Cabbie` (рис. 7.9) и соответствующими открытыми интерфейсами, любое изменение реализации `Cabbie` должно быть прозрачным для всех остальных классов. Однако в любой конструкции изменение суперкласса, безусловно, нельзя назвать прозрачным для того или иного подкласса. Понимаете, в чем проблема?

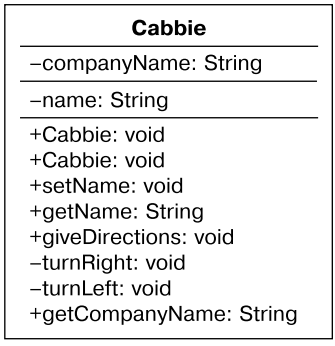

Рис. 7.9. UML-диаграмма класса Cabbie

Если бы другие классы находились в прямой зависимости от реализации класса `Cabbie`, то тестирование стало бы более сложным, а то и вовсе невозможным.

ПОСТОЯННОЕ ТЕСТИРОВАНИЕ

Даже при инкапсуляции вам потребуется повторно протестировать классы, использующие Cabbie, чтобы убедиться в том, что соответствующее изменение не привело к каким-либо проблемам.

Если вы затем создадите подкласс Cabbie с именем PartTimeCabbie, который унаследует реализацию от Cabbie, то изменение реализации Cabbie напрямую повлияет на новый класс.

Взгляните, к примеру, на UML-диаграмму, показанную на рис. 7.10. PartTimeCabbie — это подкласс Cabbie. Поэтому PartTimeCabbie наследует открытую реализацию Cabbie, включая метод giveDirections(). Если метод giveDirections() изменится в Cabbie, то это напрямую повлияет на PartTimeCabbie и все другие классы, которые позднее могут быть созданы как подклассы Cabbie. В силу этой специфики изменения реализации Cabbie не обязательно инкапсулируются в классе Cabbie.

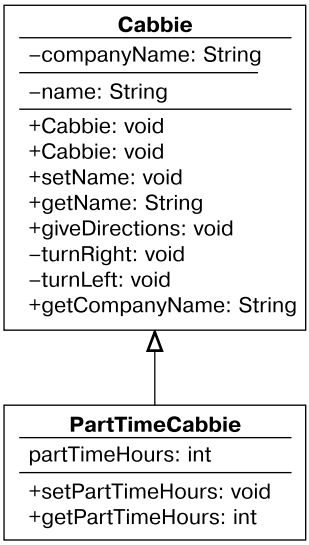

Рис. 7.10. UML-диаграмма классов Cabbie/PartTimeCabbie

Чтобы снизить риск, который представляет эта дилемма, при использовании наследования важно придерживаться строгого условия «является экземпляром». Если подкласс на самом деле является конкретизацией суперкласса, то изменения родительского класса, вероятно, подействуют на дочерний класс естественным и ожидаемым образом. Чтобы проиллюстрировать это, обратимся к следующему примеру: если класс Circle унаследует реализацию от класса Shape, а изменение реализации Shape нарушит Circle, то Circle в действительности не является конкретизацией Shape.

Как наследование может быть неправильно использовано? Рассмотрим ситуацию, когда вам требуется создать окно для целей графического интерфейса пользователя. У вас, возможно, возник бы порыв создать окно (Window), сделав его подклассом класса Rectangle:

```
public class Rectangle {

}
```

```
public class Window extends Rectangle {

}
```

На самом деле Window для графического интерфейса пользователя представляет собой нечто намного большее, чем подкласс Rectangle. Это не конкретизированная версия Rectangle, как, например, Square. Настоящий класс Window может включать Rectangle (и даже много Rectangle); вместе с тем это ненастоящий Rectangle. При таком подходе класс Window не должен наследовать от Rectangle, но должен содержать классы Rectangle:

```
public class Window {

  Rectangle menubar;
  Rectangle statusbar;
  Rectangle mainview;

}
```

Подробный пример полиморфизма

Многие люди считают полиморфизм краеугольным камнем объектно-ориентированного проектирования. Разработка класса для создания полностью независимых объектов является сутью объектно-ориентированного подхода. В хорошо спроектированной системе объект должен быть способен ответить на все важные вопросы о себе. Как правило, объект должен быть ответственным за себя. Эта независимость является одним из главных механизмов повторного использования кода.

Как уже отмечалось в главе 1, полиморфизм буквально означает *множественность форм*. При отправке сообщения объекту он должен располагать методом, позволяющим ответить на это сообщение. В иерархии наследования все подклассы наследуют интерфейсы от своих суперклассов. Однако, поскольку каждый подкласс представляет собой отдельную сущность, каждому из них может потребоваться дать отдельный ответ на одно и то же сообщение.

Повторно обратимся к примеру из главы 1, взглянув на класс Shape. Он содержит поведение Draw. Вместе с тем когда вы попросите кого-то нарисовать фигуру, первый вопрос, который вам зададут, вероятно, будет звучать так: «Какой формы?» Просто сказать человеку нарисовать фигуру будет слишком абстрактным (кстати, метод Draw в Shape не содержит реализации). Вы должны указать, фигуру какой именно формы имеете в виду. Для этого потребуется обеспечить фактическую реализацию в Circle и других подклассах. Несмотря на то что Shape содержит метод Draw, Circle переопределит этот метод и обеспечит собственный метод Draw. Переопределение, в сущности, означает замену реализации родительского класса своей собственной.

Ответственность объектов

Снова обратимся к примеру с Shape из главы 1 (рис. 7.11).

Полиморфизм — один из наиболее изящных вариантов использования наследования. Помните, что создать экземпляр Shape нельзя. Это абстрактный класс,

поскольку он содержит абстрактный метод `getArea()`. В главе 8 абстрактные классы очень подробно описаны.

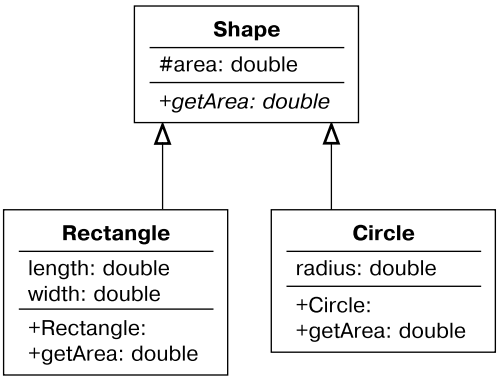

Рис. 7.11. Иерархия классов во главе с Shape

Однако экземпляры `Rectangle` и `Circle` создать можно, так как это конкретные классы. Несмотря на то что `Rectangle` и `Circle` представляют фигуры, у них имеются кое-какие различия. Поскольку речь идет о фигурах, можно вычислить их площадь. Однако формулы для вычисления площадей окажутся разными. Таким образом, формулы нельзя будет включить в класс `Shape`.

Именно здесь в дело вступает полиморфизм. Смысл полиморфизма заключается в том, что вы можете отправлять сообщения разным объектам, которые будут отвечать на них в соответствии со своими объектными типами. Например, если вы отправите сообщение `getArea()` классу `Circle`, то это приведет к вычислению с использованием формулы, отличной от той, которая будет применена, если отправить аналогичное сообщение `getArea()` классу `Rectangle`. Это потому, что `Circle` и `Rectangle` отвечают каждый за себя. Если вы попросите `Circle` возвратить значение площади круга, то он будет знать, как это сделать. Если вы захотите, чтобы `Circle` нарисовал круг, то он сможет сделать и это. Объект `Shape` не смог бы сделать этого, даже если бы можно было создать его экземпляр, поскольку у него нет достаточного количества информации о себе. Обратите внимание, что на UML-диаграмме (см. рис. 7.11) метод `getArea()` в классе `Shape` выделен курсивом. Это означает, что данный метод является абстрактным.

В качестве очень простого примера представьте, что у вас имеется четыре класса: абстрактный класс `Shape` и конкретные классы `Circle`, `Rectangle` и `Star`. Вот код:

```
public abstract class Shape{

  public abstract void draw();

}

public class Circle extends Shape{
```

```
    public void draw() {

      System.out.println("Я рисую круг");

    }
  }

  public class Rectangle extends Shape{

    public void draw() {
      System.out.println("Я рисую прямоугольник");

    }
  }

  public class Star extends Shape{

    public void draw() {

      System.out.println("Я рисую звезду");

    }
  }
```

Обратите внимание, что для каждого класса есть только один метод — `draw()`. Вот что важно для полиморфизма и объектов, которые отвечают за себя: конкретные классы сами несут ответственность за функцию рисования. Класс `Shape` не обеспечивает кода для осуществления рисования; классы `Circle`, `Rectangle` и `Star` делают это сами. Вот код как доказательство этого:

```
  public class TestShape {

    public static void main(String args[]) {

      Circle circle = new Circle();
      Rectangle rectangle = new Rectangle();
      Star star = new Star();

      circle.draw();
      rectangle.draw();
      star.draw();

    }

  }
```

Тестовое приложение `TestShape` создает три класса: `Circle`, `Rectangle` и `Star`. Чтобы нарисовать соответствующие им фигуры, `TestShape` просит отдельные классы сделать это:

```
circle.draw();
rectangle.draw();
star.draw();
```

Выполнив TestShape, вы получите следующие результаты:

```
C:\>java TestShape
Я рисую круг
Я рисую прямоугольник
Я рисую звезду
```

Это и есть полиморфизм в действии. Что бы было, если бы вы захотели создать новый класс, например Triangle? Вам потребовалось бы просто написать этот класс, скомпилировать, протестировать и использовать его. Базовому классу Shape не пришлось бы претерпевать изменения, равно как и любому другому коду:

```
public class Triangle extends Shape{

  public void draw() {

    System.out.println("Я рисую треугольник");

  }
}
```

Теперь можно отправлять сообщение Triangle. И хотя класс Shape не знает, как нарисовать треугольник, Triangle известно, как это сделать:

```
public class TestShape {

  public static void main(String args[]) {

    Circle circle = new Circle();
    Rectangle rectangle = new Rectangle();
    Star star = new Star();
    Triangle triangle = new Triangle ();

    circle.draw();
    rectangle.draw();
    star.draw();
    triangle.draw();

  }

}

C:\>java TestShape
Я рисую круг
Я рисую прямоугольник
Я рисую звезду
Я рисую треугольник
```

Чтобы увидеть истинную мощь полиморфизма, вы можете передать объект Shape методу, который абсолютно не имеет понятия, какую фигуру предстоит нарисовать. Взгляните на приведенный далее код, который включает параметры, обозначающие определенные фигуры:

```
public class TestShape {

  public static void main(String args[]) {

    Circle circle = new Circle();
    Rectangle rectangle = new Rectangle();
    Star star = new Star();

    drawMe(circle);
    drawMe(rectangle);
    drawMe(star);

  }

  static void drawMe(Shape s) {
    s.draw();
  }

}
```

В данном случае объект Shape может быть передан методу drawMe(), который способен обеспечить рисование любой допустимой фигуры — даже такой, которую вы добавите позднее. Вы можете выполнить эту версию TestShape точно так же, как и предыдущую.

Абстрактные классы, виртуальные методы и протоколы

Абстрактные классы, как они определяются на Java, также могут быть непосредственно реализованы на .NET и C++. Неудивительно, что код, написанный на C# .NET, похож на код, который написан на Java, как показано далее:

```
public abstract class Shape
{

  это конструкция, схожая с интерфейсом Java-типа,
  называемым протоколом (рассматриваемым далее public abstract void draw();

}
```

Код, написанный на Visual Basic .NET, выглядит так:

```
Public MustInherit Class Shape

  Public MustOverride Function draw()

End Class
```

Аналогичная функциональность может быть обеспечена на C++ с использованием виртуальных методов, а код будет выглядеть следующим образом:

```
class Shape
{
  public:
    virtual void draw() = 0;
}
```

Как уже отмечалось в предыдущих главах, Objective-C не полностью реализует функциональность абстрактных классов.

Например, взгляните на приведенный далее код Java-интерфейса для класса `Shape`:

```
public abstract class Shape{

  public abstract void draw();

}
```

Соответствующий протокол Objective-C показан в следующем коде. Обратите внимание, что в коде, написанном как на Java, так и на Objective-C, нет реализации для метода `draw()`:

```
@protocol Shape

@required
- (void) draw;

@end // Shape
```

На данном этапе функциональность абстрактного класса и протокола является почти одинаковой, однако именно здесь интерфейс Java-типа и протокол Objective-C различаются. Взгляните на приведенный далее Java-код:

```
public abstract class Shape{

  public abstract void draw();
  public void print() {
    System.out.println("Я осуществляю вывод");
  }

}
```

В приведенном выше примере, написанном на Java, метод `print()` обеспечивает код, который может быть унаследован тем или иным подклассом. Несмотря на то что дело обстоит аналогичным образом и в C# .NET, VB .NET и C++, этого нельзя сказать о протоколе Objective-C, который выглядел бы так:

```
@protocol Shape

@required
- (void) draw;
```

```
- (void) print;

@end // Shape
```

В этом протоколе предусмотрена подпись метода `print()`, в силу чего она должна быть реализована подклассом; вместе с тем включение кода невозможно. Коротко говоря, подклассы не могут напрямую наследовать какой-либо код от протокола, поэтому протокол нельзя использовать тем же образом, что и абстрактный класс, а это имеет значение при проектировании объектной модели.

Резюме

Эта глава содержит базовый обзор того, что представляют собой наследование и композиция и чем они отличаются. Многие авторитетные проектировщики, предпочитающие объектно-ориентированные технологии, утверждают, что композицию следует применять при наличии возможности, а наследование — только тогда, когда это необходимо.

Однако это немного упрощенный подход. Я считаю, что озвученное утверждение скрывает реальную проблему, которая может заключаться в том, что композиция является более подходящей в большем количестве случаев, чем наследование, а не в том, что ее следует использовать при наличии возможности. Тот факт, что композиция может оказаться более подходящей в большинстве случаев, не означает, что наследование — это зло. Используйте как композицию, так и наследование, но только в соответствующем контексте.

В предшествующих главах концепции абстрактных классов и Java-интерфейсов поднимались несколько раз. В главе 8 мы обратим внимание на концепцию контрактов на разработку, а также рассмотрим, как классы и Java-интерфейсы используются для выполнения этих контрактов.

Ссылки

- *Хольцнер Стивен*. Краткое наглядное руководство пользователя по Objective-C (Visual Quickstart Guide, Objective-C). — Беркли: Peachpit Press, 2010.
- *Буч Гради, Максимчук Роберт А., Энгл Майкл У., Янг Бобби Дж., Коналлен Джим и Хьюстон Келли А*. Объектно-ориентированный анализ и проектирование с примерами приложений (Object-Oriented Analysis and Design with Applications). 3-е изд. — Бостон: Addison-Wesley, 2007.
- *Майерс Скотт*. Эффективное использование C++ (Effective C++). 3-е изд. — Бостон: Addison-Wesley Professional, 2005.
- *Коуд Петер и Мейфилд Марк*. Проектирование на Java (Java Design). — Аппер-Сэддл-Ривер: Prentice-Hall, 1997.
- *Гилберт Стивен и Маккарти Билл*. Объектно-ориентированное проектирование на Java (Object-Oriented Design in Java). — Беркли: The Waite Group Press (Pearson Education), 1998.

Примеры кода, использованного в этой главе

Приведенный далее код написан на C# .NET. Эти примеры соответствуют Java-коду, продемонстрированному в текущей главе.

```
using System;

namespace TestShape
{
  public class TestShape
  {

    public static void Main()
    {

      Circle circle = new Circle();
      Rectangle rectangle = new Rectangle();

      circle.draw();
      rectangle.draw();

    }

  }

  public abstract class Shape
  {

    public abstract void draw();

  }

  public class Circle : Shape
  {

    public override void draw()
    {

      Console.WriteLine("Я рисую круг");

    }
  }

  public class Rectangle : Shape
  {

    public override void draw()
    {
```

```
        Console.WriteLine("Я рисую прямоугольник");

      }
    }

    public class Star : Shape
    {

      public override void draw()
      {

        Console.WriteLine("Я рисую звезду");

      }
    }

    public class Triangle : Shape
    {

      public override void draw()
      {

        Console.WriteLine("Я рисую треугольник");

      }
    }
  }
```

Глава 8

Фреймворки и повторное использование: проектирование с применением интерфейсов и абстрактных классов

Из главы 7 вы узнали, что наследование и композиция играют главные роли в проектировании объектно-ориентированных систем. В этой главе приведены более подробные сведения о концепциях интерфейсов Java-стиля, протоколов Objective-C и абстрактных классов.

Интерфейсы, протоколы и абстрактные классы — это мощные механизмы повторного использования кода, обеспечивающие фундамент для того, что я называю *концепцией контрактов*. В этой главе мы рассмотрим такие темы, как повторное использование кода, фреймворки, контракты, интерфейсы, протоколы и абстрактные классы (если не указано иное, я буду употреблять термин «интерфейс» в том числе и для обозначения протоколов Objective-C). В конце главы мы рассмотрим пример того, как все эти концепции могут быть применены в реальной ситуации.

Код: использовать повторно или нет?

Программисты решают вопрос повторного использования кода с тех пор, как написали свою первую строку кода. Во многих парадигмах разработки программного обеспечения повторное использование кода подчеркивается как основная часть процесса. С тех пор, когда программное обеспечение только начало появляться, концепция повторного использования кода переосмысливалась несколько раз. Объектно-ориентированная парадигма ничем не отличается в этом плане. Одно из основных преимуществ, расхваливаемых сторонниками объектно-ориентированного подхода, заключается в том, что если вы надлежащим образом напишете код изначально, то сможете использовать его повторно сколько вашей душе угодно.

Однако это правда лишь отчасти. Как и в случае со всеми подходами к проектированию, полезность кода, а также его пригодность к повторному использованию зависит от того, насколько хорошо он был спроектирован и реализован. Объектно-ориентированное проектирование «не владеет патентом» на повторное использование кода. Нет ничего, что мешает кому-либо написать очень надежный

и пригодный для повторного использования код на том или ином языке программирования, который не является объектно-ориентированным. Безусловно, несметное количество программ и функций, написанных на структурных языках вроде COBOL, C и традиционного VB, имеют высокое качество и вполне пригодны для повторного использования.

Таким образом, ясно, что приведенная далее объектно-ориентированная парадигма является не единственным способом разработки кода, пригодного для повторного использования. Однако объектно-ориентированный подход предусматривает несколько механизмов, облегчающих разработку такого кода. Один из способов создания пригодного для повторного использования кода заключается в создании фреймворков. В этой главе мы сосредоточимся на использовании интерфейсов и абстрактных классов для создания фреймворков и способствования разработке кода, пригодного для повторного использования.

Что такое фреймворк

Рука об руку с концепцией повторного использования кода идет концепция *стандартизации*, которую иногда называют концепцией *«включил и работай»*. Идея фреймворка «вращается» вокруг принципа *«включил и работай»*, а также принципа повторного использования. Один из классических примеров фреймворка — настольное приложение. Возьмем в качестве примера приложение из офисного пакета. В редакторе документов, которым я пользуюсь на данный момент (Microsoft Word 2010), имеется лента, включающая разные вкладки. Эти вкладки подобны тем, что есть в презентационном пакете (Microsoft PowerPoint 2010) и программном обеспечении для работы с электронными таблицами (Microsoft Excel 2010), которые тоже сейчас открыты у меня. Фактически первые два элемента меню (Главная, Вставка) одинаковы во всех трех программах. Пункты меню схожи, кроме того, многие из подменю тоже подобны (Создать, Открыть, Сохранить и т. д.). Под лентой находится область документа — будь то документ, презентация или электронная таблица. Общий фреймворк облегчает освоение различных приложений, содержащихся в офисном пакете. В то же время он облегчает жизнь разработчикам, позволяя по максимуму повторно использовать код, а также части того, что было спроектировано ранее.

То, что все эти строки меню выглядят схожими, явно не случайно. Фактически при проектировании в большинстве интегрированных сред разработки на конкретной платформе, например Microsoft Windows, вы получаете определенные вещи без необходимости самим создавать их. При создании окна в среде Windows у вас автоматически появятся такие элементы, как строка основного заголовка и кнопка закрытия файлов, находящаяся в правом верхнем углу. Действия тоже стандартизированы: когда вы дважды щелкаете кнопкой мыши на строке основного заголовка, окно всегда сворачивается/разворачивается. Когда вы нажимаете кнопку закрытия в правом верхнем углу, выполнение приложения всегда завершается. Все это является частью фреймворка. На рис. 8.1 приведен скриншот приложения Microsoft Word. Обратите внимание на строки меню, панели инструментов и прочие элементы, которые являются частью фреймворка.

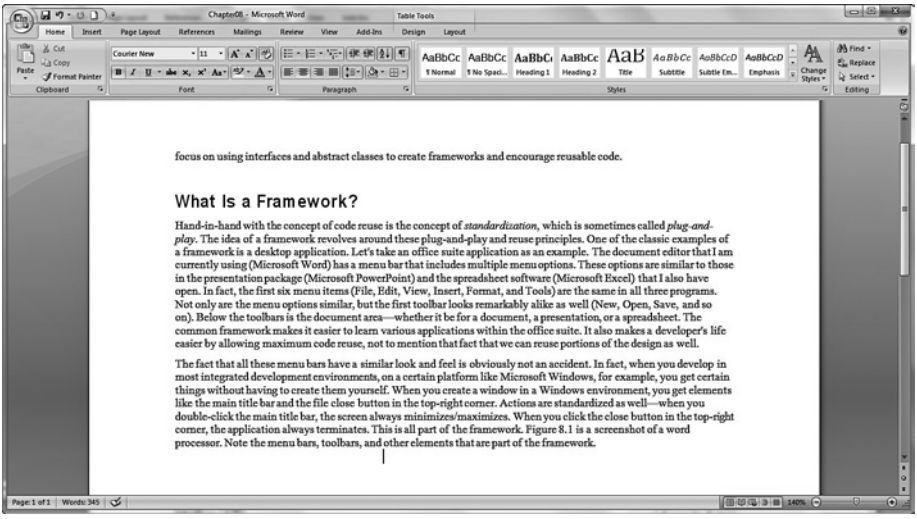

Рис. 8.1. Фреймворк для обработки текста

Фреймворк для обработки текста, как правило, позволяет выполнять такие операции, как создание, открытие и сохранение документов, удаление, копирование и вставка текста, поиск в документах и т. д. Для того чтобы можно было воспользоваться этим фреймворком, разработчик должен прибегнуть к предопределенному интерфейсу для создания приложения. Этот интерфейс соответствует стандартному фреймворку, у которого имеется два очевидных преимущества. Во-первых, как мы видели ранее, внешний вид является единообразным и конечным пользователям не придется осваивать новый фреймворк. Во-вторых, разработчик сможет воспользоваться преимуществами кода, уже написанного и протестированного (а то, что тестирование кода уже было проведено, является огромным преимуществом). Зачем писать код для создания совершенно нового диалогового окна Открыть, если он уже есть и был тщательно протестирован? В сфере бизнеса, где время имеет решающее значение, людям не хочется, чтобы им приходилось осваивать новые вещи, если только это не является абсолютно необходимым.

ЕЩЕ РАЗ О ПОВТОРНОМ ИСПОЛЬЗОВАНИИ

В главе 7 мы говорили о повторном использовании кода в том плане, в каком оно касается наследования, — по сути один класс наследует от другого. Эта глава посвящена фреймворкам и повторному использованию целых или частичных систем.

Сам собой напрашивается следующий вопрос: если вам понадобится диалоговое окно, то каким образом вы будете использовать диалоговое окно, обеспечиваемое фреймворком? Ответ прост: вам потребуется следовать правилам, которые предусмотрены для соответствующего фреймворка. А где эти правила можно найти? Правила, касающиеся фреймворка, можно найти в документации. Человек или люди, написавшие класс, классы или библиотеки классов, должны были обеспечить документацию по использованию открытых интерфейсов класса, классов

или библиотек классов (по крайней мере, мы надеемся на это). Во многих случаях все это имеет форму интерфейса программирования приложений (API — Application Programming Interface).

Например, для того, чтобы создать строку меню на Java, вам потребовалось бы воспользоваться API-документацией к классу `JMenuBar` и взглянуть на открытые интерфейсы, которые там представлены. На рис. 8.2 показана часть Java API. Используя такие API-интерфейсы, вы сможете создавать полноценные Java-апплеты, придерживаясь при этом требуемых стандартов. Если вы будете соблюдать эти стандарты, то ваши апплеты смогут работать в браузерах с включенной поддержкой Java.

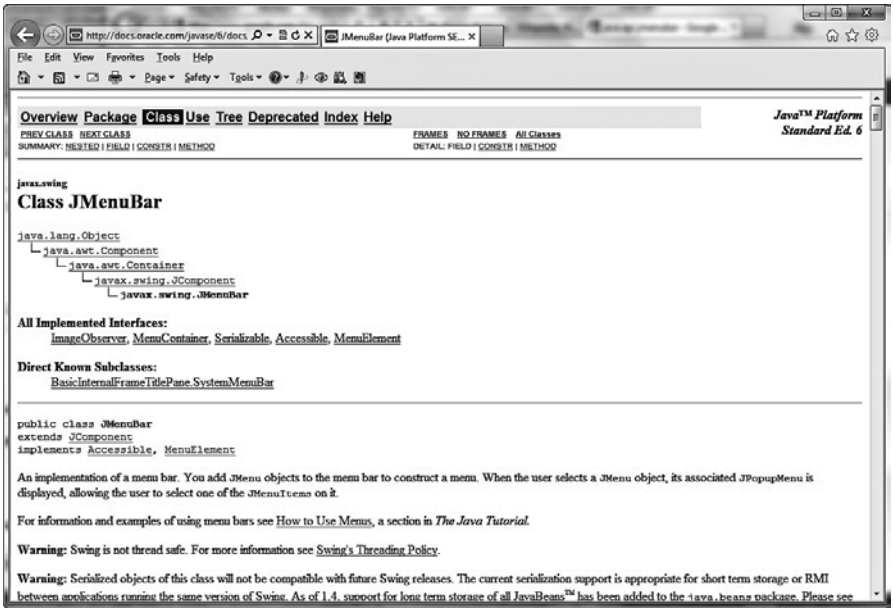

Рис. 8.2. API-документация

Что такое контракт

В контексте этой главы мы будем рассматривать *контракт* как любой механизм, требующий от разработчиков соблюдения спецификаций того или иного API-интерфейса. Часто бывает так, что API-интерфейс называют фреймворком. Онлайн-словарь Dictionary.com (http://www.dictionary.com) определяет контракт как «соглашение между двумя и более сторонами о совершении или несовершении неких оговоренных действий», а также как «соглашение, которое может быть принудительно выполнено по закону».

Именно это и происходит, когда разработчик использует API-интерфейс, — менеджер проекта, частный предприниматель или отраслевой стандарт при этом обеспечивает принудительное следование требуемым нормам. При использовании

контрактов разработчик обязан соблюдать правила, определенные для фреймворка. Сюда входят имена методов, количество параметров и т. д. (подписи и т. п.). Коротко говоря, стандарты создаются с целью способствовать использованию правильных методик разработки.

ТЕРМИН «КОНТРАКТ»

Термин «контракт» широко используется во многих областях бизнеса, включая разработку программного обеспечения. Не путайте представленную здесь концепцию с другими возможными концепциями проектирования программного обеспечения, которые тоже называются контрактами.

Принудительное следование требуемым нормам жизненно важно, поскольку всегда существует вероятность того, что разработчик нарушит контракт. Без принудительного следования требуемым нормам жуликоватый разработчик может решить «изобрести велосипед» и написать код по-своему, вместо того чтобы придерживаться спецификации, предусмотренной для определенного фреймворка. От стандарта мало пользы, если люди игнорируют или обходят его. При использовании таких языков программирования, как Java и .NET, два способа реализации контрактов заключаются в применении абстрактных классов и интерфейсов соответственно.

Абстрактные классы

Один из способов реализации контракта состоит в использовании абстрактного класса. *Абстрактный класс* — это класс, содержащий один или несколько методов, которые не имеют какой-либо обеспеченной реализации. Допустим, у вас есть абстрактный класс `Shape`. Он является абстрактным потому, что нельзя создать его экземпляр. Если вы попросите кого-нибудь нарисовать фигуру, то первый вопрос, который вам зададут, скорее всего, будет звучать так: «Какой формы?» Таким образом, концепция фигуры является абстрактной. Однако если кто-нибудь попросит вас нарисовать круг, то в этом случае проблема будет не совсем такой же, поскольку круг является конкретной концепцией. Вы знаете, как выглядит круг. Вы также знаете, как нарисовать фигуры других форм, например прямоугольники. Как все это применимо к контрактам? Предположим, вам требуется создать приложение для рисования фигур. Наша цель заключается в рисовании всевозможных фигур, представленных в текущей конструкции, а также тех, что могут быть добавлены позднее. Есть два условия, которых мы должны придерживаться в данном случае.

Во-первых, нам необходимо, чтобы для рисования всех фигур использовался один и тот же синтаксис. Например, мы хотим, чтобы любой класс, который представляет ту или иную фигуру и реализован в нашей системе, содержал метод с именем `draw()`. Таким образом, опытные разработчики будут косвенно знать, что для рисования той или иной фигуры вы вызовете метод `draw()`, независимо от того, какой она будет формы. Теоретически это сокращает количество времени, затрачиваемого на копание в руководствах, а также способствует снижению количества синтаксических ошибок.

Во-вторых, важно, чтобы каждый класс отвечал за свои действия. Таким образом, несмотря на необходимость того, чтобы в классе был предусмотрен метод с именем `draw()`, этот класс должен обеспечивать собственную реализацию кода. Например, в обоих классах `Circle` и `Rectangle` будет присутствовать метод `draw()`, однако очевидно, что в классе `Circle` будет иметься код для рисования кругов, а в `Rectangle`, как и следовало ожидать, будет содержаться код для рисования прямоугольников. Когда мы в конечном счете создадим классы с именами `Circle` и `Rectangle`, которые будут подклассами `Shape`, им потребуется реализовать собственную версию `Draw` (рис. 8.3).

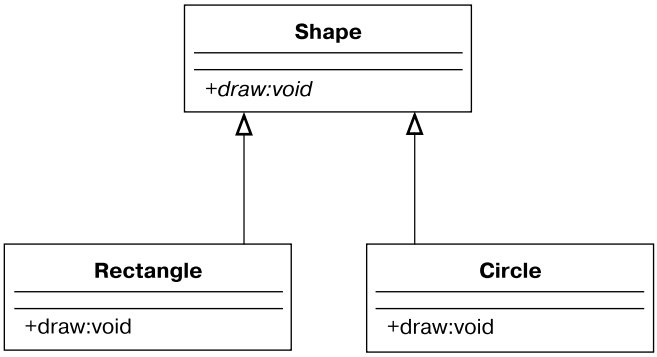

Рис. 8.3. Иерархия абстрактных классов

Таким образом, у нас есть фреймворк, который является по-настоящему полиморфным. Метод `Draw` может вызываться для рисования любой фигуры, предусмотренной в системе, однако его вызов приводит к разным результатам. Вызов метода `Draw` из объекта `Circle` приводит к рисованию круга, а вызов метода `Draw` из объекта `Rectangle` — к рисованию прямоугольника. В сущности, отправка сообщения объекту вызывает разную ответную реакцию в зависимости от того, каким именно является этот объект. В этом и заключается суть полиморфизма:

```
circle.draw();      // рисует круг
rectangle.draw();   // рисует прямоугольник
```

Взглянем на код, который показывает, как `Rectangle` и `Circle` соблюдают контракт с `Shape`. Вот код для класса `Shape`:

```
public abstract class Shape {

  public abstract void draw();  // реализация отсутствует

}
```

Обратите внимание, что класс не обеспечивает какой-либо реализации для `draw()`; по сути код отсутствует, что делает метод `draw()` абстрактным (обеспечение кода сделало бы этот метод конкретным). Отсутствие реализации объясняется двумя причинами. Во-первых, `Shape` не знает, что рисовать, в силу чего мы не смогли бы реализовать метод `draw()`, даже если бы захотели.

СТРУКТУРНАЯ АНАЛОГИЯ

Это интересный момент. Если бы мы пожелали, чтобы класс Shape содержал код для рисования всевозможных фигур, как нынешних, так и будущих, то нам потребовался бы условный оператор (вроде case). Тогда сопровождение всего кода оказалось бы очень запутанным и сложным. Это лишь один пример того, где будут кстати преимущества объектно-ориентированного проектирования.

Во-вторых, нам нужно, чтобы подклассы обеспечивали реализацию. Взглянем на классы `Circle` и `Rectangle`:

```
public class Circle extends Shape {

  public void Draw() {System.out.println ("Рисование круга")};

}

public class Rectangle extends Shape {

  public void Draw() {System.out.println ("Рисование прямоугольника")};

}
```

Обратите внимание, что оба класса `Circle` и `Rectangle` расширяют (то есть наследуют от) `Shape`. Заметьте также, что они обеспечивают фактическую реализацию. Именно здесь в дело вступает контракт. Если `Circle` будет наследовать от `Shape`, однако в итоге в нем не окажется метода `draw()`, то `Circle` не пройдет даже компиляцию. Таким образом, `Circle` не сможет выполнить контракт с `Shape`. Менеджер проекта может потребовать, чтобы программисты, создающие для приложения классы, которые представляют фигуры, обеспечили наследование от `Shape`. Благодаря этому все такие классы в приложении будут содержать метод `draw()`, который станет работать ожидаемым образом.

CIRCLE

Если у Circle действительно не получится реализовать метод draw(), то этот класс будет считаться абстрактным. Таким образом, еще один подкласс должен наследовать от Circle и реализовать метод draw(). Тогда этот подкласс станет конкретной реализацией обоих — Shape и Circle.

Хотя концепция абстрактных классов «вращается» вокруг абстрактных методов, ничто не мешает `Shape` обеспечить реализацию (помните, что определение абстрактного класса заключается в том, что он содержит *один* или *несколько* абстрактных методов, — это означает, что абстрактный класс также может включать конкретные методы). Например, несмотря на то что `Circle` и `Rectangle` по-разному реализуют метод `draw()`, они совместно используют один и тот же механизм для задания цвета фигуры. Таким образом, класс `Shape` может содержать атрибут `color` и метод для задания цвета. Метод `setColor()` является конкретной реализацией и был бы унаследован обоими `Circle` и `Rectangle`. Единственными методами, которые подкласс должен реализовать, являются те, что объявлены в суперклассе абстрактными. Эти абстрактные методы представляют собой контракт.

ПРЕДОСТЕРЕЖЕНИЕ

Знайте, что в случае с Shape, Circle и Rectangle мы имеем дело с отношением строгого наследования, а не с отношением интерфейса, о котором пойдет речь в следующем разделе. Circle является экземпляром Shape так же, как и Rectangle. Это важно, поскольку контракты не используются в случае с отношениями композиции или «содержит как часть».

В некоторых языках программирования, например C++, для реализации контрактов используются только абстрактные классы; вместе с тем Java и .NET располагают другим механизмом, который реализует контракт, называемый *интерфейсом*. В прочих языках программирования, например Objective-C, абстрактные классы не предусмотрены. Таким образом, для того чтобы реализовать контракт при работе с Objective-C, вам потребуется использовать протокол, который является версией интерфейса Objective-C.

Интерфейсы

Перед тем как определять интерфейс, интересно отметить, что в C++ нет конструкции с таким названием. Применяя C++, вы, в принципе, можете создать интерфейс, использовав синтаксическое подмножество абстрактного класса. Например, приведенный далее код на C++ является абстрактным классом. Однако поскольку единственный метод в этом классе — виртуальный, реализация отсутствует. В результате этот абстрактный класс обеспечивает ту же функциональность, что и интерфейс.

```
class Shape
{
  public:
    virtual void draw() = 0;
}
```

ТЕРМИНОЛОГИЯ, СВЯЗАННАЯ С ИНТЕРФЕЙСАМИ

Это еще один из тех случаев, когда программная терминология оказывается запутанной, даже очень запутанной. Знайте, что термин «интерфейс» можно использовать в нескольких значениях.

Первый: графический интерфейс пользователя (GUI — Graphical User Interface) широко применяется для обозначения визуального интерфейса, с которым взаимодействует пользователь, зачастую — на мониторе.

Второй: интерфейс для класса — это, в сущности, подписи его методов.

Третий: при использовании Objective-C вы можете разбивать код на физически раздельные модули, называемые интерфейсом и реализацией.

Четвертый: интерфейс Java-стиля и протокол Objective-C по сути представляют собой контракт между родительским и дочерним классами.

Сам собой напрашивается следующий вопрос: если абстрактный класс может обеспечивать ту же функциональность, что и интерфейс, то зачем в Java и .NET вообще предусмотрена конструкция, называемая интерфейсом? И зачем в Objective-C предусмотрен протокол?

Прежде всего, C++ поддерживает множественное наследование, в отличие от Java, Objective-C и .NET. Несмотря на то что классы Java, Objective-C и .NET могут наследовать только от одного родительского класса, они могут реализовывать много интерфейсов. Использование нескольких абстрактных классов лежит в основе множественного наследования; таким образом, в случае применения Java и .NET нельзя пойти этим путем. Коротко говоря, при использовании интерфейса вам не придется беспокоиться о формальной структуре наследования — теоретически вы сможете добавить интерфейс в любой класс, если это будет иметь смысл при проектировании. Однако абстрактный класс требует, чтобы наследование осуществлялось от него и, соответственно, от его потенциальных родительских классов.

ИНТЕРФЕЙСЫ И МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ

По этим соображениям интерфейсы часто считают «обходными путями», компенсирующими отсутствие множественного наследования. Технически это неверно. Интерфейсы представляют собой отдельную методику проектирования, и, несмотря на то что их можно использовать для проектирования приложений с применением множественного наследования, они не являются заменой множественного наследования или «обходными путями», компенсирующими его отсутствие.

Как и абстрактные классы, интерфейсы — это мощный инструмент приведения контрактов в исполнение в случае с фреймворками. Прежде чем мы перейдем к каким-либо концептуальным определениям, будет полезно взглянуть на UML-диаграмму фактического интерфейса и соответствующий код. Посмотрите на интерфейс `Nameable`, показанный на рис. 8.4.

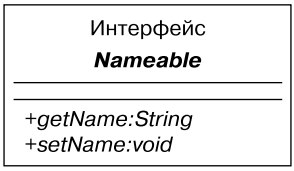

Рис. 8.4. UML-диаграмма Java-интерфейса

Обратите внимание, что `Nameable` определен на UML-диаграмме как интерфейс, благодаря чему его можно отличить от обычного класса (абстрактного или нет). Заметьте также, что интерфейс содержит два метода: `getName()` и `setName()`. Вот соответствующий код:

```
public interface Nameable {

  String getName();
  void setName (String aName);

}
```

Вот для сравнения код соответствующего протокола Objective-C:

```
@protocol Nameable

@required
```

```
    - (char *) getName;
    - (void) setName: (char *) n;
@end // Nameable
```

Обратите внимание, что в этом коде `Nameable` объявлен не как класс, а как интерфейс. Из-за этого методы `getName()` и `setName()` считаются абстрактными, а реализация отсутствует. Интерфейс, в отличие от абстрактного класса, может не обеспечивать вообще *никакой* реализации. В результате любой класс, реализующий интерфейс, должен обеспечивать реализацию для всех методов. Например, в Java класс наследует от абстрактного класса, в то время как класс реализует интерфейс.

НАСЛЕДОВАНИЕ РЕАЛИЗАЦИИ И НАСЛЕДОВАНИЕ ОПРЕДЕЛЕНИЯ

Наследование иногда называют наследованием реализации, а интерфейсы — наследованием определения.

Связываем все воедино

Если и абстрактные классы, и интерфейсы содержат абстрактные методы, то в чем заключается реальная разница между ними? Как мы уже видели ранее, абстрактные классы включают абстрактные и конкретные методы, а интерфейсы содержат только абстрактные методы. Почему же они так отличаются в этом плане?

Допустим, нам необходимо спроектировать класс, представляющий собаку, с таким расчетом, что мы будем позднее добавлять и другие классы млекопитающих. Логическим ходом в данной ситуации было бы создание абстрактного класса `Mammal`.

```
public abstract class Mammal {

  public void generateHeat() {System.out.println("Выработка тепла");}

  public abstract void makeNoise();

}
```

Этот класс содержит конкретный метод `generateHeat()` и абстрактный метод `makeNoise()`. Первый является конкретным, поскольку все млекопитающие вырабатывают тепло. Второй является абстрактным, потому что все млекопитающие издают разные звуки.

Создадим также класс `Head`, который будем использовать, когда речь зайдет об отношении композиции:

```
public class Head {

  String size;

  public String getSize() {

    return size;

  }
```

```
    public void setSize(String aSize) { size = aSize;}

}
```

Класс `Head` содержит два метода — `getSize()` и `setSize()`. Несмотря на то что композиция, может, и объясняет разницу между абстрактными классами и интерфейсами, ее использование в этом примере иллюстрирует то, как она связана с абстрактными классами и интерфейсами в общей конструкции объектно-ориентированной системы. Я считаю, что это важно, поскольку так пример выглядит полным. Помните, что есть два способа установления объектных отношений: использование отношения «является экземпляром», представляемого наследованием, и применение отношения «содержит как часть», представляемого композицией. Вопрос состоит в следующем: куда именно «вписывается» интерфейс?

Чтобы ответить на этот вопрос и связать все воедино, создадим класс с именем `Dog`, который будет подклассом `Mammal`, реализующим `Nameable` и обладающим объектом `Head` (рис. 8.5).

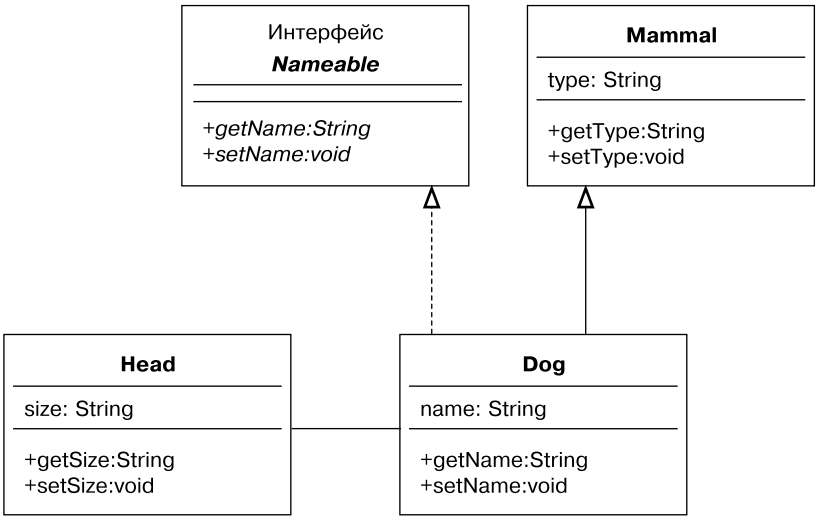

Рис. 8.5. UML-диаграмма образца кода

Если говорить кратко, то Java и .NET позволяют создавать объекты тремя путями — с помощью наследования, интерфейсов и композиции. Обратите внимание, что штриховая линия на рис. 8.5 представляет интерфейс. Этот пример показывает, когда вам следует использовать каждую из этих конструкций. Когда выбирать абстрактный класс? Когда выбирать интерфейс? Когда выбирать композицию? Разберемся подробнее.

Вам должны быть уже знакомы следующие концепции:

- `Dog` является подклассом `Mammal`, поэтому здесь имеет место отношение наследования;

- Dog реализует Nameable, поэтому здесь имеет место отношение интерфейса;
- Dog обладает Head, поэтому здесь имеет место отношение композиции.

Приведенный далее код демонстрирует, как можно включить абстрактный класс и интерфейс в один и тот же класс:

```
public class Dog extends Mammal implements Nameable {

  String name;

  Head head;

  public void makeNoise(){System.out.println("Лай");}

  public void setName (String aName) {name = aName;}
  public String getName () {return (name);}

}
```

После того как вы взглянете на UML-диаграмму, у вас может возникнуть вопрос: хотя штриховая линия от Dog к Nameable и представляет интерфейс, разве это все же не наследование? На первый взгляд ответ не так прост. Несмотря на то что интерфейсы — это особый тип наследования, важно знать, что именно означает слово «особый» в данном случае. Понимание *особой* разницы является ключом к грамотному объектно-ориентированному проектированию.

Хотя наследование представляет собой строгое отношение «является экземпляром», это не совсем так в случае с интерфейсом. Рассмотрим такой пример.

- Собака является млекопитающим.
- Рептилия не является млекопитающим.

Таким образом, класс Reptile не смог бы наследовать от класса Mammal. Однако интерфейс выходит за пределы разных классов.

- Собаке можно дать имя.
- Ящерице можно дать имя.

Ключ здесь в том, что классы при строгом наследовании должны быть связанными. Например, в рассматриваемой нами конструкции класс Dog находится в непосредственной связи с классом Mammal. Собака является млекопитающим. Собаки и ящерицы не связаны на уровне млекопитающих, поскольку ящерица — это не млекопитающее.

Однако интерфейсы могут использоваться для классов, которые не являются связанными. Вы можете дать имя собаке так же, как и ящерице. В этом и заключается ключевая разница между использованием абстрактного класса и интерфейса.

Абстрактный класс представляет некоторую реализацию. Фактически мы видели, что в Mammal содержится конкретный метод generateHeat(). Даже если мы не знаем, с каким млекопитающим имеем дело, нам все равно известно, что все млекопитающие вырабатывают тепло. Однако интерфейс моделирует только поведение. Интерфейс *никогда* не обеспечивает реализации какого-либо рода — только поведение. Он определяет поведение, которое будет одинаковым во всех классах,

между которыми, возможно, не окажется никакой связи. Имена можно давать не только собакам, но и машинам, планетам и т. д.

Код, выдерживающий проверку компилятором

Можно ли доказать или опровергнуть, что в случае с интерфейсами имеет место настоящее отношение «является экземпляром»? В ситуации с Java (это также может быть сделано при использовании C# и VB) мы можем позволить компилятору выяснить все за нас. Взгляните на приведенный далее код:

```
Dog D = new Dog();
Head H = D;
```

Если прогнать этот код через компилятор, то будет выведено следующее сообщение об ошибке:

```
Test.java:6: Несовместимый тип идентификатора. Не могу преобразовать Dog
в Head. Head H = D;
```

Ясно, что `Dog` — это не `Head`. Мы знаем это, и компилятор согласен. Однако, как и следовало ожидать, с приведенным далее кодом все будет отлично:

```
Dog D = new Dog();
Mammal M = D;
```

Это настоящее отношение наследования, и неудивительно, что компилятор разбирает этот код с положительным результатом, поскольку `Dog` является подклассом `Mammal`.

Теперь мы можем по-настоящему протестировать интерфейс. Представляет ли он собой настоящее отношение «является экземпляром»? Компилятор считает, что да:

```
Dog D = new Dog();
Nameable N = D;
```

С этим кодом все отлично. Таким образом, мы можем с уверенностью сказать, что экземпляр класса `Dog` — это сущность, которой можно дать имя. Это простое, но эффективное доказательство того, что как наследование, так и интерфейсы представляют собой отношение «является экземпляром».

ИНТЕРФЕЙС NAMEABLE

Интерфейс определяет конкретное поведение, а не реализацию. Реализуя интерфейс Nameable, вы подразумеваете, что обеспечите соответствующее поведение с помощью реализации методов getName() и setName(). Вам решать, как именно вы это сделаете. Все, что вам потребуется, — обеспечить данные методы.

Заключение контракта

Простое правило при определении контракта заключается в обеспечении нереализованного метода с помощью либо абстрактного класса, либо интерфейса. Таким образом, когда подкласс проектируется с намерением реализовать контракт, он должен обеспечивать реализацию нереализованных методов в родительском классе или интерфейсе.

Как уже отмечалось, одно из преимуществ контрактов — стандартизация соглашений по программированию. Подробнее рассмотрим эту концепцию, взглянув на пример того, что бывает, когда не используются стандарты программирования. В данном случае у нас будет три класса: `Planet`, `Car` и `Dog`. Каждый из них реализует код для задания имени сущности. Однако, поскольку все они реализованы по отдельности, каждый класс располагает отличающимся синтаксисом для извлечения имени. Взгляните на приведенный далее код для класса `Planet`:

```
public class Planet {

  String planetName;

  public void getplanetName() {return planetName;};

}
```

Аналогичным образом класс `Car` мог бы иметь такой код:

```
public class Car {

  String carName;

  public String getCarName() { return carName;};

}
```

А класс `Dog` мог бы иметь следующий код:

```
public class Dog {

  String dogName;

  public String getDogName() { return dogName;};

}
```

Очевидно здесь то, что любому, кто воспользуется этими классами, придется заглянуть в документацию (какая ужасная мысль!) для того, чтобы выяснить, как извлечь имя в каждом из этих случаев. Хотя необходимость заглянуть в документацию — не самое худшее, что может случиться, было бы здорово, если бы для всех классов, используемых в проекте (или компании), применялось одно и то же соглашение об именовании — это немного облегчило бы жизнь. Именно здесь в дело вступает интерфейс `Nameable`.

Идея состоит в том, чтобы заключить контракт, охватывающий классы любых типов, которым требуются имена. По мере того как пользователи разных классов будут переходить от одного класса к другому, им не придется выяснять текущий синтаксис для именования объекта. Все классы `Planet`, `Car` и `Dog` станут задействовать один и тот же синтаксис именования.

Чтобы достичь этой высокой цели, мы можем создать интерфейс (у нас есть возможность воспользоваться интерфейсом `Nameable`, который мы применяли ранее). Суть соответствующего соглашения заключается в том, что все классы

должны реализовывать Nameable. Таким образом, пользователям придется запомнить только один интерфейс для всех классов в том, что касается соглашений об именовании:

```
public interface Nameable {

  public String getName();
  public void setName(String aName);

}
```

Новые классы Planet, Car и Dog должны выглядеть так:

```
public class Planet implements Nameable {

  String planetName;

  public String getName() {return planetName;}
  public void setName(String myName) { planetName = myName;}

}

public class Car implements Nameable {

  String carName;

  public String getName() {return carName;}
  public void setName(String myName) { carName = myName;}

}

public class Dog implements Nameable {

  String dogName;

  public String getName() {return dogName;}
  public void setName(String myName) { dogName = myName;}

}
```

В данном случае у нас имеется стандартный интерфейс, при этом мы использовали контракт для гарантии того, что дело будет обстоять именно так. Фактически одним из главных преимуществ применения той или иной современной интегрированной среды разработки является то, что при реализации интерфейса она будет автоматически обеспечивать заглушки требуемых методов. Эта функция позволяет сэкономить много времени и сил при использовании интерфейсов.

Есть одна небольшая проблема, о которой вы, возможно, задумывались. Идея контракта великолепна, если все играют по правилам, но что если какая-нибудь сомнительная личность не захочет соблюдать правила (например, жуликоватый программист)? Суть в том, что людям ничто не мешает нарушить стандартный контракт, однако в таких случаях они столкнутся с большими проблемами.

С одной стороны, менеджер проекта может настоять на том, чтобы все соблюдали контракт, точно так же, как и на том, что все члены команды должны использовать одни и те же соглашения об именовании переменных и систему управления конфигурацией. Если тот или иной член команды не станет соблюдать правила, то ему могут сделать выговор или даже уволить.

Обеспечение следования правилам — один из способов гарантировать, что контракты соблюдаются. При этом бывают ситуации, когда результатом нарушения контракта оказывается непригодный к использованию код. Возьмем, к примеру, Java-интерфейс `Runnable`. Java-апплеты реализуют интерфейс `Runnable`, поскольку он требует, чтобы любой реализующий его класс обязательно реализовывал метод `run()`. Это важно, так как браузер, вызывающий апплет, будет вызывать метод `run()`, содержащийся в `Runnable`. Если метод `run()` будет отсутствовать, то произойдет ошибка.

Системные «точки расширения»

По сути контракты являются «точками расширения» в вашем коде. Везде, где вам нужно сделать части системы абстрактными, вы можете использовать контракт. Вместо того чтобы создавать связи с объектами определенных классов, можно «подключиться» к любому объекту, реализующему контракт. Вам необходимо знать, где именно контракты окажутся полезны; вместе с тем возможно и злоупотребление контрактами. Вам потребуется выявить общие детали вроде интерфейса `Nameable`, о котором мы говорили в этой главе. Но знайте, что в случае применения контрактов возможны компромиссные решения. Они могут сделать возможность повторного использования кода более реальной, однако несколько усложняют работу.

Пример из сферы электронного бизнеса

Иногда трудно убедить человека, который принимает решения и не имеет никакого опыта разработки, в том, что повторное использование кода позволяет сэкономить финансовые средства. Однако при повторном использовании кода довольно легко понять преимущества этого подхода. В текущем разделе мы подробно рассмотрим простой, но практический пример того, как создать работоспособный фреймворк с применением наследования, абстрактных классов, интерфейсов и композиции.

Проблема, касающаяся электронного бизнеса

Пожалуй, наилучший способ понять всю мощь повторного использования — рассмотреть пример того, как оно может осуществляться. В этом примере мы прибегнем к наследованию (с помощью интерфейсов и абстрактных классов) и композиции. Наша цель — создать фреймворк, который сделает повторное использование кода реальностью, сократит время, уходящее на написание кода, и упростит сопровождение, то есть претворит в жизнь все то, что входит в типичный список пожеланий при разработке программного обеспечения.

Откроем наш собственный интернет-бизнес. Предположим, что у нас есть клиент — небольшая пиццерия под названием Papa's Pizza. Хотя это небольшое семейное предприятие, его владелец осознает, что присутствие в Интернете может во многих отношениях помочь бизнесу. Он хочет, чтобы клиенты смогли зайти на его сайт, узнать, что такое Papa's Pizza, и удобно заказать пиццу прямо из своих браузеров.

На сайт, который мы будем разрабатывать, клиенты смогут зайти, выбрать там продукцию, которую они желают заказать, и указать вариант и время доставки. Они также смогут съесть заказанное в ресторане, забрать свой заказ самостоятельно либо получить его через курьера. Например, клиенту в 15:00 захочется заказать на ужин пиццу (с салатами, хлебными палочками и напитками), которая должна быть доставлена к нему домой в 18:00. Допустим, этот клиент находится на работе (и у него, конечно же, сейчас перерыв). Он заходит на соответствующий сайт и выбирает пиццу, указывая, какого она должна быть размера, чем должна быть покрыта и должна ли у нее быть корочка. Затем он выбирает салаты, включая приправы, а также хлебные палочки и напитки. Далее клиент выбирает вариант доставки и просит, чтобы его заказ был доставлен к нему домой в 18:00. После этого он оплачивает заказ кредитной карточкой, получает номер подтверждения и выходит из системы. Через несколько минут он также получает подтверждение по электронной почте. Мы создадим систему учетных записей, благодаря чему люди, снова зашедшие на этот сайт, увидят приветствие с их именами, которое также будет содержать информацию о том, какая пицца у них любимая и какие новые виды пиццы готовятся на этой неделе.

Когда программная система в конце концов будет готова, это ознаменует громкий успех. На протяжении последующих нескольких недель клиенты Papa's Pizza будут с радостью заказывать пиццу и другие блюда с напитками через Интернет. Допустим, во время этого периода раскрутки родственник главы Papa's Pizza, владеющий магазином пончиков под названием Dad's Donuts, наносит ему визит. Глава Papa's Pizza показывает владельцу Dad's Donuts свою систему, которая последнему очень нравится. На следующий день владелец Dad's Donuts звонит в нашу компанию и просит разработать веб-систему для его магазина пончиков. Это здорово, и это именно то, на что мы надеялись. Как мы теперь можем выгодно использовать код, который задействовали при создании системы для пиццерии, чтобы создать систему для магазина пончиков?

Сколько еще небольших предприятий, помимо Papa's Pizza и Dad's Donuts, смогли бы воспользоваться преимуществами нашего фреймворка, для того чтобы преуспеть в Интернете? Если у нас получится разработать хороший, надежный фреймворк, то мы сможем успешно создавать веб-системы, которые будут дешевле тех, что мы были в состоянии создавать прежде. Кроме того, есть и дополнительное преимущество: код будет уже протестирован и реализован, благодаря чему отладка и сопровождение должны стать намного проще.

Подход без повторного использования кода

По многим причинам концепция повторного использования кода не настолько успешна, насколько хотелось бы некоторым разработчикам программного обеспечения.

Во-первых, при разработке систем повторное использование кода во многих случаях даже не принимается во внимание. Во-вторых, даже если оно и «входит в уравнение», такие вещи, как сжатые сроки графика, ограниченные ресурсы и бюджетные вопросы, часто становятся препятствием для самых лучших побуждений.

Во многих ситуациях код в итоге получается тесно связанным с тем приложением, для которого он был написан. Это означает, что код в приложении сильно зависит от другого кода в том же приложении.

Повторное использование кода часто оказывается результатом операций вырезания, копирования и вставки. Пока одно приложение открыто в текстовом редакторе, вы копируете код, а затем вставляете его в другое приложение. Иногда определенные функции или программы используются без внесения в них каких-либо изменений. К сожалению, нередко бывает так, что, даже если большая часть кода и сможет остаться прежней, все равно придется немного изменить его для того, чтобы он смог работать в определенном приложении.

Взгляните, к примеру, на два отдельных приложения, представленных на UML-диаграмме на рис. 8.6.

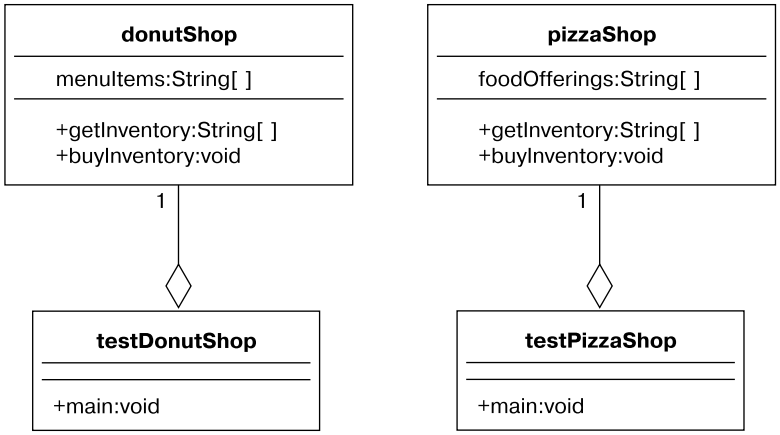

Рис. 8.6. Приложения, пути которых расходятся

В этом примере приложения `testDonutShop` и `testPizzaShop` являются полностью независимыми модулями кода. Код каждого из них хранится отдельно, и никакого взаимодействия между этими модулями нет. Однако у этих приложений может быть кое-какой общий код. Фактически часть кода могла бы быть дословно скопирована из одного приложения в другое. В определенный момент кто-либо, вовлеченный в проект, мог бы решить создать библиотеку таких совместно используемых фрагментов кода, чтобы использовать их в этих и других приложениях. Во многих хорошо организованных проектах такой подход хорошо работает. Использование стандартов программирования, управление конфигурациями, управление изменениями и т. д. очень хорошо налажены. Однако во многих случаях порядок нарушается.

Любому, кто знаком с процессом разработки программного обеспечения, известно, что, когда неожиданно обнаруживаются дефекты, а время имеет существенное значение, возникает соблазн внести в систему кое-какие исправления или дополнения, специфичные для приложения, которое в них нуждается в данный момент. Это может решить проблему для одного приложения, но также способно непреднамеренно, возможно, пагубно сказаться на других. Таким образом, в подобных ситуациях изначально совместно используемый код может разниться и приходится сопровождать отдельные кодовые базы.

Представим, например, что однажды сайт Papa's Pizza перестал работать. Его владелец в панике звонит нам, и один из наших разработчиков может отследить проблему. Разработчик устраняет проблему, зная при этом, что внесенное исправление поможет, но не вполне уверен почему. Этот разработчик также делает копию кода строго для использования в системе для Papa's Pizza. Она будет ласково называться «Версия 2.01papa». Поскольку разработчик еще не полностью понимает проблему, а также потому, что система Dad's Donuts и так отлично работает, перенос кода в систему для магазина пончиков не осуществляется.

ОТСЛЕЖИВАНИЕ ДЕФЕКТА

Тот факт, что в системе для Papa's Pizza оказался дефект, не означает, что он также будет и в системе для Dad's Donuts. Хотя этот дефект привел к тому, что сайт Papa's Pizza перестал работать, с сайтом Dad's Donuts этого может вообще никогда не случиться. Возможно, исправление кода, как в системе для Papa's Pizza, окажется более опасным для Dad's Donuts, чем первоначальный дефект.

На следующей неделе владелец Papa's Pizza снова в панике звонит, но уже с совершенной другой проблемой. Разработчик решает ее, опять-таки не зная при этом, как внесенное исправление повлияет на остальную часть системы, делает отдельную копию кода и называет ее «Версия 2.03dad». Этот сценарий разыгрывается для всех сайтов, которые сейчас находятся у нас в разработке. Теперь у нас дюжина или более копий кода с разными версиями для разных сайтов. Все это превращается в неразбериху. У нас имеется множество ветвей кода, и мы пересекли точку невозврата. Возможно, нам никогда не удастся объединить их снова (пожалуй, мы все же смогли бы, однако с точки зрения бизнеса это обошлось бы дорого).

Наша цель состоит в том, чтобы избежать путаницы, как в приведенном ранее примере. Несмотря на то что во многих системах приходится решать проблемы, связанные с унаследованным кодом, к счастью для нас, приложения для Papa's Pizza и Dad's Donuts являются совершенно новыми системами. Поэтому мы можем проявить некоторую дальновидность и спроектировать требуемую систему, прибегнув к подходу, предполагающему повторное использование кода. Таким образом, мы избежим проблем сопровождения, описанных совсем недавно. Для этого нам потребуется выделить как можно большую общность. При проектировании мы сосредоточимся на всех общих бизнес-функциях, присутствующих в веб-приложении. Вместо того чтобы располагать разными классами приложений вроде `testPizzaShop` и `testDonutShop`, мы можем создать конструкцию, включающую класс `Shop`, который будет использоваться всеми приложениями.

Обратите внимание, что у testPizzaShop и testDonutShop имеются одинаковые интерфейсы getInventory() и buyInventory(). Мы выделим эту общность и сделаем так, чтобы все приложения, соответствующие нашему фреймворку Shop, обязательно реализовывали методы getInventory() и buyInventory(). Это требование соответствия стандарту иногда называют контрактом. Четко изложив контракт об оказании услуг, вы изолируете код от одной реализации. В Java реализация контракта осуществляется с помощью интерфейса или абстрактного класса. Посмотрим, как это делается.

Решение для электронного бизнеса

Взглянем, как использовать контракт для выявления общности этих систем. В данном случае мы создадим абстрактный класс для выявления общности в реализациях, а также интерфейс (уже знакомый нам Nameable) для выявления общности в поведениях.

Наша цель — создать специальные версии нашего веб-приложения со следующими функциями:

- интерфейсом Nameable, который будет частью контракта;
- абстрактным классом Shop, который тоже будет частью контракта;
- классом CustList, который мы используем при композиции;
- новой реализацией Shop для каждого клиента, которому будут предоставляться услуги.

Объектная модель UML

Новый созданный класс Shop будет хранилищем функциональности. Обратите внимание на рис. 8.7: методы getInventory() и buyInventory() были «подняты» по дереву иерархии из DonutShop и PizzaShop в абстрактный класс Shop. Теперь всякий раз, когда нам понадобится создать новую, специальную версию Shop, мы станем добавлять новую реализацию Shop (например, GroceryShop). Shop — это контракт, которого должны придерживаться реализации:

```
public abstract class Shop {

  CustList customerList;

  public void CalculateSaleTax() {

    System.out.println("Вычисление налога на продажу");

  }

  public abstract String[] getInventory();

  public abstract void buyInventory(String item);

}
```

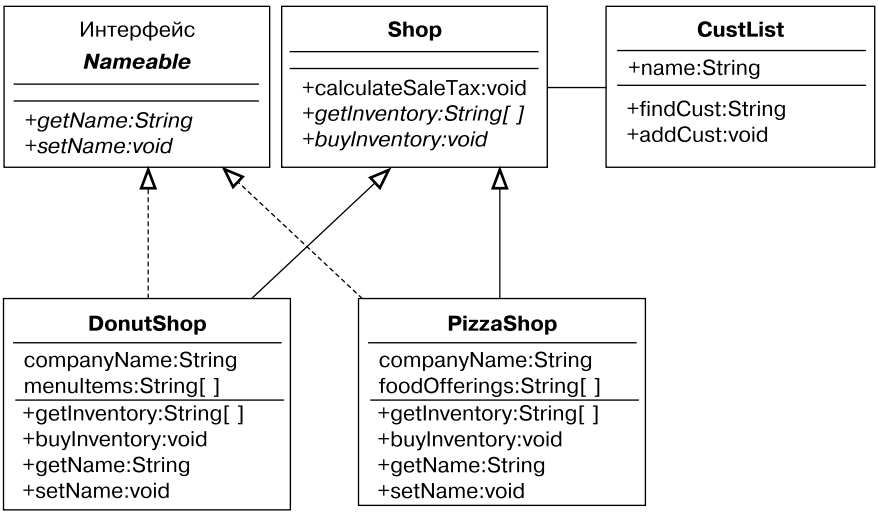

Рис. 8.7. UML-диаграмма модели Shop

Чтобы показать, как композиция вписывается в эту картину, класс `Shop` включает `CustList`. Таким образом, класс `CustList` содержится в `Shop`:

```
public class CustList {

  String name;

  public String findCust() {return name;}
  public void addCust(String Name){}

}
```

Чтобы проиллюстрировать использование интерфейса в этом примере, определяется интерфейс `Nameable`:

```
public interface Nameable {

  public abstract String getName();
  public abstract void setName(String name);

}
```

Мы потенциально могли бы располагать большим количеством разных реализаций, однако весь остальной код (приложение) был бы неизменным. В этом маленьком примере экономия кода может не показаться большой. Однако в крупном, реальном приложении экономия кода будет значительной. Взглянем на реализацию `DonutShop`:

```
public class DonutShop extends Shop implements Nameable {

  String companyName;
```

```
    String[] menuItems = {
      "Пончики",
      "Маффины",
      "Пирожное из слоеного теста",
      "Кофе",
      "Чай"
  }

    public String[] getInventory() {

      return menuItems;

    }

    public void buyInventory(String item) {

      System.out.println("\nВы только что приобрели" + item);

    }

    public String getName(){

      return companyName;
    }

    public void setName(String name){

      companyName = name;
    }
  }
```

Реализация `PizzaShop` выглядит похожим образом:

```
  public class PizzaShop extends Shop implements Nameable {

    String companyName;


    String[] foodOfferings = {
      "Пицца",
      "Спагетти",
      "Овощной салат",
      "Антипасто",
      "Кальцоне"
    }

    public String[] getInventory() {

      return foodOfferings;

    }
```

```
    public void buyInventory(String item) {

      System.out.println("\nВы только что приобрели " + item);

    }

    public String getName(){

      return companyName;
    }

    public void setName(String name){

      companyName = name;
    }

  }
```

В отличие от изначальной ситуации, когда присутствовало большое количество специальных приложений, теперь у нас имеется только один первичный класс (`Shop`) и разные специальные классы (`PizzaShop`, `DonutShop`). Связанность приложения с каким-либо из специальных классов отсутствует. Приложение связано лишь с контрактом (`Shop`). Он определяет, что любая реализация `Shop` должна обеспечивать реализацию для двух методов — `getInventory()` и `buyInventory()`. Она также должна обеспечивать реализацию для `getName()` и `setName()`, которая связана с реализуемым интерфейсом `Nameable`.

Несмотря на то что такой подход решает проблему тесно связанных реализаций, нам все еще нужно решить, какую реализацию использовать. При текущей стратегии нам все же пришлось бы располагать отдельными приложениями. По сути придется предусмотреть по одному приложению для каждой реализации `Shop`. Несмотря на использование нами контракта `Shop`, мы все равно находимся в такой же ситуации, что и раньше, когда не использовали этот контракт:

```
DonutShop myShop= new DonutShop();

PizzaShop myShop = new PizzaShop ();
```

Как же нам решить эту проблему? У нас есть возможность создавать объекты динамически. Используя Java, мы можем написать такой код:

```
String className = args[0];

Shop myShop;

myShop = (Shop)Class.forName(className).newInstance();
```

В данном случае значение для `className` задается после передачи параметра соответствующему коду (есть и другие способы задания значения для `className`, например использование системного свойства).

Взглянем на Shop с применением этого подхода (обратите внимание, что обработка исключений отсутствует и нет ничего другого, кроме создания экземпляра объекта).

```
class TestShop {

  public static void main (String args[]) {

    Shop shop = null;

    String className = args[0];

    System.out.println("Создание экземпляра класса:" + className + "\n");

    try {

      // new pizzaShop();
      shop = (Shop)Class.forName(className).newInstance();

    } catch (Exception e) {

      e.printStackTrace();
    }

    String[] inventory = shop.getInventory();
    // показ списка товаров

    for (int i=0; i<inventory.length; i++) {
      System.out.println("Аргумент" + i + " = " + inventory[i]);
    }

    // покупка товара

    shop.buyInventory(Inventory[1]);

  }

}
```

Таким образом, мы можем использовать один и тот же программный код как для PizzaShop, так и для DonutShop. Если мы добавим приложение GroceryShop, то нам потребуется лишь обеспечить реализацию и соответствующую строку в основном приложении. Изменять программный код не понадобится.

Резюме

При проектировании классов и объектных моделей жизненно важно понимать, как объекты связаны друг с другом. В этой главе мы рассмотрели основные вопросы создания объектов — наследование, интерфейсы и композицию. Из нее вы узнали, как создавать пригодный для повторного использования код путем проектирования с применением контрактов.

В главе 9 мы завершим наше объектно-ориентированное «путешествие» и исследуем, как объекты, которые могут быть абсолютно несвязанными, способны взаимодействовать друг с другом.

Ссылки

- *Хольцнер Стивен*. Краткое наглядное руководство пользователя по Objective-C (Visual Quickstart Guide, Objective-C). — Беркли: Peachpit Press, 2010.
- *Буч Гради, Максимчук Роберт А., Энгл Майкл У., Янг Бобби Дж., Коналлен Джим и Хьюстон Келли А.* Объектно-ориентированный анализ и проектирование с примерами приложений (Object-Oriented Analysis and Design with Applications). 3-е изд. — Бостон: Addison-Wesley, 2007.
- *Майерс Скотт*. Эффективное использование C++ (Effective C++). 3-е изд. — Бостон: Addison-Wesley Professional, 2005.
- *Коуд Петер и Мейфилд Марк*. Проектирование на Java (Java Design). — Аппер-Сэддл-Ривер: Prentice-Hall, 1997.

Примеры кода, использованного в этой главе

Приведенный далее код написан на C# .NET. Эти примеры соответствуют Java-коду, продемонстрированному в текущей главе.

Пример TestShop: C# .NET

```
using System;

namespace TestShop
{
  class TestShop
  {

    public static void Main()
    {

      Shop shop = null;

      Console.WriteLine("Создание экземпляра класса PizzaShop:" + "\n");

      shop = new PizzaShop();

      string[] inventory = shop.getInventory();

      // показ списка товаров

      for (int i = 0; i < 5; i++)
      {
        Console.WriteLine("Аргумент" + i + " = " + inventory[i]);
      }
```

```
    // покупка товара

    shop.buyInventory(inventory[1]);

  }

}


public abstract class Shop {

  public void CalculateSaleTax() {

    Console.WriteLine("Вычисление налога на продажу");

  }
  public abstract string[] getInventory();

  public abstract void buyInventory(string item);

}

public interface Nameable {

  string getName();
  void setName(string name);

}

public class PizzaShop : Shop , Nameable
{

  string _CompanyName;

  string[] foodOfferings = {
    "Пицца",
    "Спагетти",
    "Овощной салат",
    "Антипасто",
    "Кальцоне"
  }

  public override string[] getInventory() {

    return foodOfferings;

  }

  public override void buyInventory(string item) {
```

```
        Console.WriteLine("\nВы только что приобрели " + item);

      }

      public string getName(){

        return _CompanyName;
      }

      public void setName(string name){

        _CompanyName = name;
      }

    }

    public class DonutShop : Shop , Nameable {

      string _CompanyName;

      string[] menuItems = {
        "Пончики",
        "Маффины",
        "Пирожное из слоеного теста",
        "Кофе",
        "Чай"
      }

      public override string[] getInventory() {

        return menuItems;

      }

      public override void buyInventory(string item) {

        Console.WriteLine(string.format("\nВы только что приобрели{0}.", item);

      }

      public string getName(){

        return _CompanyName;
      }

      public void setName(string name){

        _CompanyName = name;
      }
    }
  }
```

Глава 9

Создание объектов и объектно-ориентированное проектирование

В двух предыдущих главах мы рассмотрели темы наследования и композиции. Из главы 7 вы узнали, что наследование и композиция — это основные способы создания объектов. А из главы 8 вам стало известно, что существуют разные степени наследования, а также то, как наследование, интерфейсы, абстрактные классы и композиция сочетаются друг с другом.

В этой главе рассматривается, как объекты связаны друг с другом в общей конструкции. Вы могли бы сказать, что эта тема уже была разобрана ранее, и оказались бы правы. И наследование, и композиция — способы взаимодействия объектов. Однако между ними есть одно существенное различие в плане подхода к созданию объектов. При использовании наследования конечным результатом, по крайней мере концептуально, является класс, который включает все поведения и атрибуты иерархии наследования. А при использовании композиции для создания нового класса применяется один или несколько классов.

Хотя наследование представляет собой отношение между двумя классами, на самом деле при использовании этого механизма создается родительский класс, который включает атрибуты и методы дочернего класса. Снова обратимся к примеру классов `Person` и `Employee` (рис. 9.1).

Несмотря на то что в данном случае действительно имеется два отдельно спроектированных класса, отношение между ними нельзя назвать просто взаимодействием — это отношение наследования. По сути `Employee` представляет собой `Person`. Объекту `Employee` не нужно отправлять сообщение объекту `Person`. Объект `Employee` не нуждается в услугах `Person`, ведь объект `Employee` — это объект `Person`.

Однако в случае с композицией дело обстоит иначе. Композиция — это взаимодействие между разными объектами. Таким образом, в то время как в главе 8 были рассмотрены преимущественно разные типы наследования, в этой главе мы углубимся в различные типы композиции и разберем, как объекты взаимодействуют друг с другом.

Отношения композиции

Мы уже видели ранее: композиция означает, что тот или иной элемент является частью некоего целого. Отношение наследования выражается как отношение «яв-

ляется экземпляром», а композиция — как отношение «содержит как часть». Мы интуитивно знаем, что автомобиль содержит как часть руль (рис. 9.2).

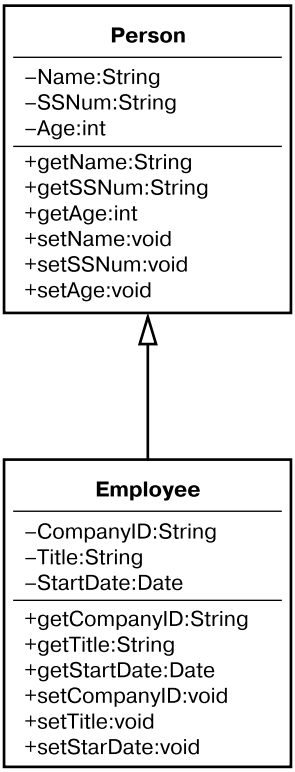

Рис. 9.1. Отношение наследования

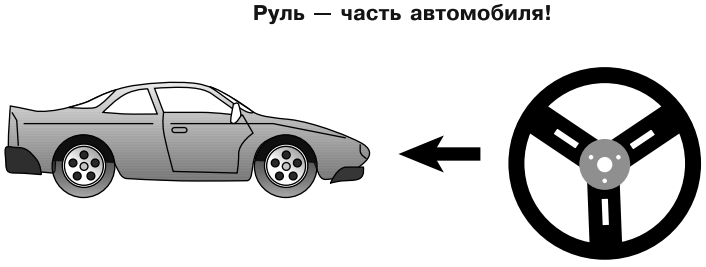

Рис. 9.2. Отношение композиции

Композицию следует использовать потому, что она позволяет создавать системы путем объединения менее сложных частей. Это распространенный среди людей подход к рассмотрению проблем. Исследования показывают, что даже наиболее способные из нас могут одновременно удержать в кратковременной памяти максимум семь порций данных. Поэтому нам нравится использовать абстрактные

концепции. Вместо того чтобы говорить, что у нас есть большое устройство с рулем, четырьмя покрышками, двигателем и т. д., мы говорим, что у нас есть автомобиль. Так нам легче изъясняться и сохранять ясность в своей голове.

Композиция также помогает и другим образом, например, в том, чтобы сделать части взаимозаменяемыми. Если бы все рули были одинаковыми, то не имело бы значения, какой конкретно руль устанавливается в конкретном автомобиле. В сфере разработки программного обеспечения взаимозаменяемые части подразумевают повторное использование.

В главах 7 и 8 своей книги под названием «Объектно-ориентированное проектирование на Java» *(Object-Oriented Design in Java)* Стивен Гилберт и Билл Маккарти приводят большое количество подробных примеров ассоциаций и композиции. Я настоятельно рекомендую вам обратиться к этому материалу для более глубокого взгляда на соответствующие вопросы. А здесь мы рассмотрим некоторые существенные особенности этих концепций и исследуем несколько вариаций их примеров.

Поэтапное создание

Еще одно основное преимущество использования композиции состоит в том, что системы и подсистемы можно создавать независимо и, пожалуй, что более важно, тестировать и сопровождать независимо.

Нет сомнения, что программные системы довольно сложны. Чтобы создать качественное программное обеспечение, вы должны придерживаться одного важнейшего правила, которое позволит вам добиться успеха: все нужно делать максимально простым. Чтобы большие программные системы работали должным образом и были легки в сопровождении, их следует разделить на менее крупные, более управляемые части. Как это сделать? В 1962 году в статье под названием «Архитектура сложности» *(The Architecture of Complexity)* лауреат Нобелевской премии Герберт Саймон (Herbert Simon) изложил следующие мысли относительно стабильных систем.

- **«Стабильные сложные системы обычно представлены в форме иерархии, где любая система состоит из более простых подсистем, каждая из которых тоже состоит из более простых подсистем»** — вам, возможно, уже знаком этот принцип, поскольку он лежит в основе функциональной декомпозиции — метода, стоящего за процедурной разработкой программного обеспечения. При объектно-ориентированном проектировании аналогичные принципы распространяются и на композицию — создание сложных объектов из более простых частей.
- **«Стабильные сложные системы почти не поддаются декомпозиции»** — это означает, что вы можете идентифицировать части, образующие систему, и отличить взаимодействия между частями и внутри частей. В стабильных системах меньше связей между их частями, чем внутри их частей. Таким образом, модульная стереосистема с простыми связями между звуковыми колонками, проигрывателем и усилителем по своей природе более стабильна, чем интегрированная система, декомпозиция которой не является легкой.
- **«Стабильные сложные системы почти всегда состоят из подсистем лишь нескольких разных типов, упорядоченных в разных комбинациях»** — эти

подсистемы, в свою очередь, обычно состоят из частей лишь нескольких разных типов.

- **«Стабильные сложные системы почти всегда развиваются из простых рабочих систем»** — вместо того чтобы создавать новую систему с нуля, то есть изобретать велосипед, в качестве ее основы следует использовать проверенные конструкции, которые ей предшествуют.

Допустим, в нашем примере стереосистема (рис. 9.3) является полностью интегрированной и не разделенной на образующие ее компоненты (то есть представляет собой систему в виде одного большого черного ящика). Что бы было, если бы CD-плеер сломался и стал непригодным к использованию? Вам пришлось бы нести в ремонт всю систему целиком. Это оказалось бы сложнее и дороже, кроме того, вы не смогли бы пользоваться другими компонентами.

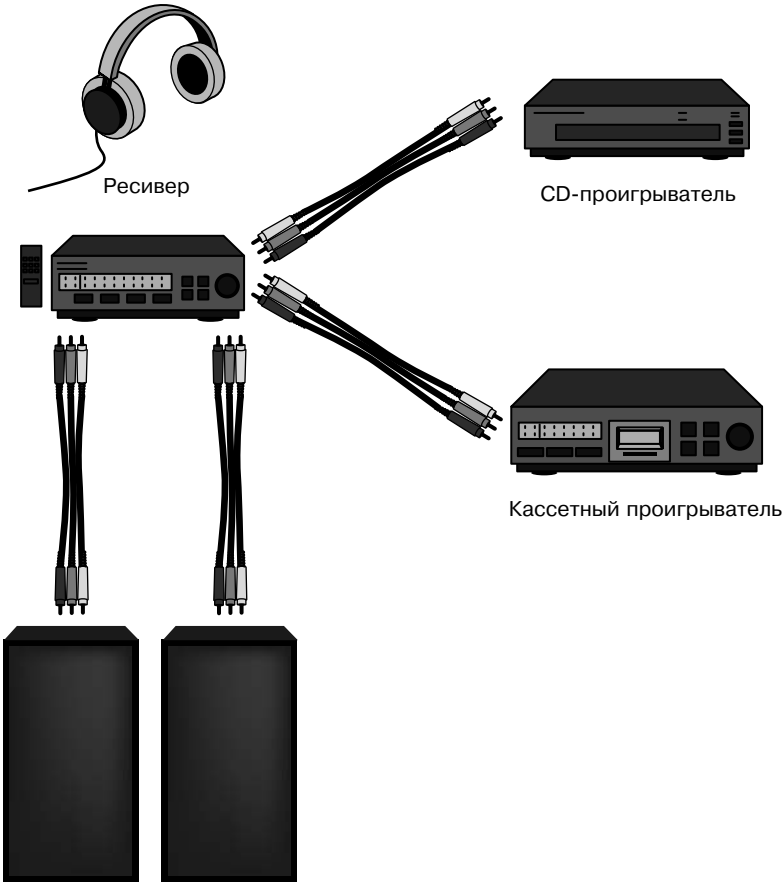

Рис. 9.3. Поэтапное создание, тестирование и верификация полной системы

Эта концепция становится очень важной для языков программирования вроде Java и тех, что включены в фреймворк .NET. Поскольку объекты загружаются динамически,

разделение конструкции является очень важным. Например, если вы распределите Java-приложение и при этом понадобится воссоздать один из файлов классов (для устранения ошибок или сопровождения), то вам придется перераспределить только этот конкретный файл класса. Если бы весь код располагался в одном файле, то потребовалось бы перераспределять все приложение целиком.

Допустим, система разделена на компоненты, а не является единым блоком. Если при этом сломается CD-плеер, то вы сможете отсоединить его и отнести в ремонт (заметьте, что все компоненты связаны соединительными шнурами). Это будет легче и дешевле, а также займет меньше времени, чем если бы вам пришлось возиться с единым, интегрированным блоком. Дополнительное преимущество состоит в том, что вы все равно сможете пользоваться остальной частью системы. Вы даже сможете купить новый CD-плеер, поскольку он является компонентом. В то же время мастер сможет подключить ваш сломанный CD-плеер к своей ремонтной системе, чтобы проверить его и починить. В целом компонентный подход работает довольно хорошо. Композиция — это одна из основных стратегий, которые имеются в арсенале у вас как разработчика программного обеспечения, и позволяют бороться с его сложностью.

Одно из основных преимуществ использования компонентов заключается в том, что вы можете задействовать компоненты, созданные другими разработчиками или даже сторонними поставщиками. Однако применение того или иного компонента из другого источника требует определенной степени доверия к нему. Сторонние компоненты должны происходить из надежного источника, и вы должны быть уверены в том, что это программное обеспечение было протестировано, не говоря уже о том, что оно должно как следует выполнять заявленные функции. По-прежнему существует много таких людей, которые предпочитают создать свои собственные компоненты, нежели доверять тем, что были созданы другими.

Типы композиции

В целом существует два типа композиции — ассоциация и агрегация. В обоих случаях отношения представляют собой взаимодействия между объектами. В примере со стереосистемой, который только что использовался для объяснения одного из основных преимуществ композиции, была продемонстрирована ассоциация.

ЯВЛЯЕТСЯ ЛИ КОМПОЗИЦИЯ ФОРМОЙ АССОЦИАЦИИ?

Композиция — это еще одна область в объектно-ориентированных технологиях, где имеет место вопрос «Что было раньше — курица или яйцо?». В одних учебниках говорится, что композиция является формой ассоциации, а в других — что ассоциация является формой композиции. Так или иначе, в этой книге мы считаем наследование и композицию двумя основными способами создания классов. Таким образом, в этой книге ассоциация считается формой композиции.

Все формы композиции включают отношение «содержит как часть». Однако между ассоциациями и агрегациями имеются тонкие различия, которые зависят от того, как вы представляете себе части целого. В случае с агрегациями вы обычно видите только целое, а в случае с ассоциациями — части, которые образуют целое.

Агрегации

Пожалуй, наиболее интуитивно понятной формой композиции является агрегация. Она означает, что сложный объект состоит из других объектов. Телевизор представляет собой ясный и точный пример устройства, которое вы используете для развлечения. Глядя на свой телевизор, вы видите один телевизор. Большую часть времени вы не перестаете думать о том, что в состав телевизора входят микрочипы, экран, тюнер и т. д. Естественно, вы видите переключатель для включения/выключения телевизора и, конечно же, экран. Однако люди обычно не так представляют себе телевизоры. Когда вы приходите в магазин бытовой техники, продавец не говорит: «Позвольте показать вам эту агрегацию микрочипов, экрана, тюнера и т. д.». Он говорит: «Позвольте показать вам этот телевизор».

Аналогичным образом, когда вы отправляетесь покупать автомобиль, вы не планируете выбирать все его отдельные компоненты. Вы не собираетесь решать, какие свечи зажигания или дверные ручки купить. Вы идете покупать автомобиль. Разумеется, вы все же выберете некоторые его функции, но по большей части будете выбирать автомобиль как целое, сложный объект, состоящий из множества других сложных и простых объектов (рис. 9.4).

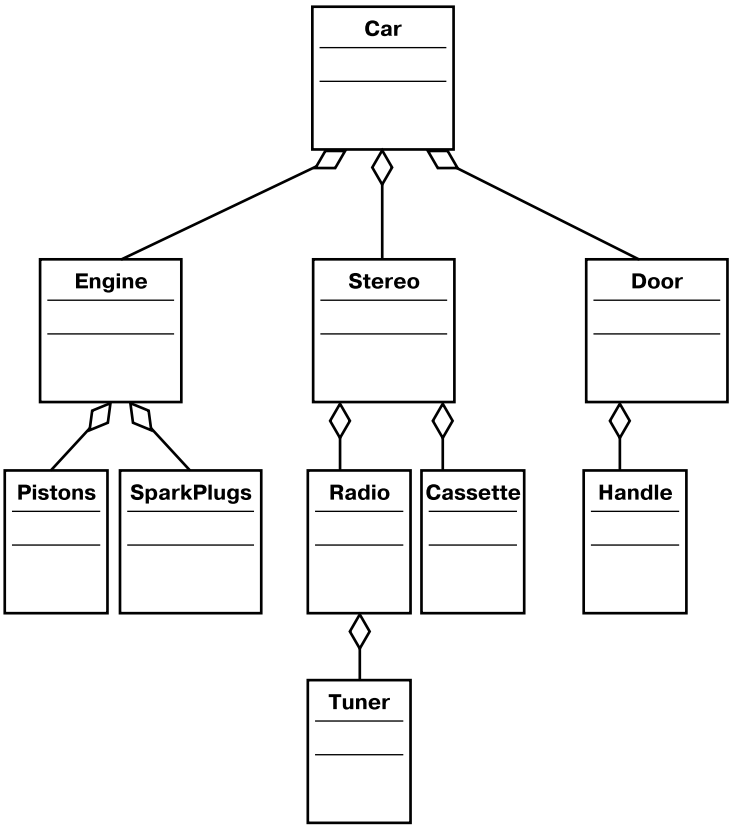

Рис. 9.4. Иерархия агрегаций для Car

Ассоциации

В то время как агрегации представляют отношения, при которых вы обычно видите целое, ассоциации представляют как целое, так и части. Как уже отмечалось в примере со стереосистемой, разные компоненты располагаются по отдельности и подключаются к целому соединительными шнурами (которые связывают разные компоненты).

В качестве примера взгляните на компьютерную систему (рис. 9.5). Как вы понимаете, компьютерная система — это целое. Компонентами являются монитор, клавиатура, мышь и системный блок компьютера. Каждый из них — это отдельный объект, однако все вместе они образуют целую компьютерную систему. Системный блок использует клавиатуру, мышь и монитор, чтобы поручить им некоторую часть работы. Другими словами, системный блок компьютера нуждается в услугах мыши и при этом не способен предоставить такую услугу сам. Поэтому он запрашивает соответствующую услугу у отдельной мыши через определенный порт и кабель, соединяющий мышь с этим системным блоком компьютера.

АГРЕГАЦИЯ В ПРОТИВОПОСТАВЛЕНИИ С АССОЦИАЦИЕЙ

Агрегация — это сложный объект, состоящий из других объектов. Ассоциация используется, когда одному объекту нужно, чтобы другой объект оказал ему услугу.

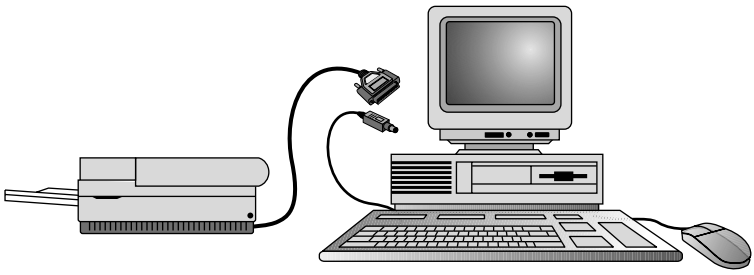

Рис. 9.5. Ассоциации как отдельная услуга

Использование ассоциаций в сочетании с агрегациями

Во всех этих примерах вы, возможно, заметили одну вещь: разделительная линия между тем, что такое ассоциация, и тем, что такое агрегация, часто оказывается размытой. Достаточно сказать, что многие из ваших наиболее интересных проектных решений будут сводиться к выбору того, использовать ассоциации или же агрегации.

Например, образец компьютерной системы, приводившийся ранее для описания ассоциаций, также включает агрегацию. Несмотря на то что взаимодействие между системным блоком компьютера, монитором, клавиатурой и мышью является ассоциацией, системный блок компьютера как таковой представляет собой агрегацию. Вы видите только системный блок компьютера, но на самом деле это сложная система, состоящая из других объектов, включая чипы, материнскую плату, видеокарту и т. д.

Допустим, объект `Employee` состоит из объектов `Address` и `Spouse`. Вы, возможно, посчитаете, что объект `Address` является агрегацией (по сути, частью объекта `Employee`), а объект `Spouse` — ассоциацией. В целях пояснения предположим, что оба — `Employee` и `Spouse` — представляют работников. Если соответствующий работник будет уволен, то `Spouse` все равно останется в системе, однако произойдет нарушение ассоциации.

Аналогичным образом в примере со стереосистемой у ресивера имеется ассоциация со звуковыми колонками, а также с CD-плеером. Кроме того, звуковые колонки и CD-плеер сами по себе являются агрегациями других объектов, как, например, силовые кабели.

Хотя в примере с автомобилем двигатель, свечи зажигания и двери представляют композицию, стереосистема также представляет отношение ассоциации.

ОДНОГО ПРАВИЛЬНОГО ОТВЕТА НЕ СУЩЕСТВУЕТ

Как и обычно, нет какого-то одного абсолютно правильного ответа, когда дело касается принятия проектного решения. Проектирование не является точной наукой. Хотя мы можем устанавливать общие правила, чтобы жить по ним, они не являются жесткими.

Избегание зависимостей

При использовании композиции желательно не делать объекты сильно зависящими друг от друга. Один из способов сделать объекты сильно зависящими друг от друга заключается в смешении доменов. Объект в одном домене не должен смешиваться с объектом в другом домене за исключением определенных ситуаций. Вернемся к примеру со стереосистемой, чтобы разобраться в этой концепции.

Если ресивер и CD-плеер будут «располагаться» в отдельных доменах, то стереосистему будет легче поддерживать в работоспособном состоянии. Например, если сломается такой компонент, как CD-плеер, то вы сможете отправить его в ремонт отдельно. В данном случае CD-плеер и MP3-плеер «обладают» отдельными доменами. Это обеспечивает гибкость, например возможность купить CD- и MP3-плеер от разных производителей. Таким образом, если вы решите заменить CD-плеер устройством от другого производителя, то сможете это сделать.

Иногда смешение доменов несет в себе определенное удобство. Хороший пример этого: существование комбинаций «телевизор/видеомагнитофон» и «телевизор/DVD-плеер». С момента выхода первого издания этой книги эволюционировало не только объектно-ориентированное проектирование, но и потребительские технологии. Видеомагнитофоны уже не так популярны, как прежде, а DVD-проигрыватели следуют тем же путем. Смена предпочтений потребителей с видеомагнитофонов на DVD-плееры, а затем на потоковые технологии за столь непродолжительный период времени — это хороший пример того, почему нужно избегать зависимостей при проектировании на многих уровнях.

Вам необходимо решить, что важнее при определенных обстоятельствах: удобство или стабильность. Одного правильного ответа не существует. Все зависит от приложения и среды. В случае с комбинацией «телевизор/видеомагнитофон» мы решили, что удобство интегрированного блока значительно перевешивало риск его более низкой стабильности (рис. 9.6).

Рис. 9.6. Удобство в противопоставлении со стабильностью

СМЕШЕНИЕ ДОМЕНОВ

Важно понять, обеспечит ли смешение доменов удобство. Если преимущества обладания комбинацией «телевизор/видеомагнитофон» перевешивают риск и потенциальное время простоя отдельных компонентов, то смешение доменов может оказаться предпочтительным выбором при проектировании.

Кардинальность

В своей книге под названием «Объектно-ориентированное проектирование на Java» (*Object-Oriented Design in Java)* Гилберт и Маккарти описывают кардинальность как количество объектов, участвующих в ассоциации, с указанием того, является это участие обязательным или необязательным. Чтобы определить кардинальность, Гилберт и Маккарти ставят следующие вопросы.

- Какие именно объекты будут взаимодействовать с какими именно другими объектами?
- Сколько объектов будет принимать участие при каждом взаимодействии?
- Взаимодействие будет обязательным или необязательным?

К примеру, взглянем на следующий образец. Мы создадим класс `Employee`, который будет наследовать от `Person` и иметь отношения с приведенными далее классами:

- `Division`;
- `JobDescription`;
- `Spouse`;
- `Child`.

Что эти классы делают? Являются ли они необязательными? Сколько их требуется `Employee`?

- `Division`.
  - Этот объект содержит информацию, касающуюся отдела, в котором трудится работник.

- Каждый работник должен трудиться в отделе, поэтому отношение является обязательным.
- Работник трудится в одном и только одном отделе.

- `JobDescription`.
  - Этот объект содержит сведения о служебных обязанностях, в том числе, скорее всего, такую информацию, как шкала заработной платы и диапазон уровня заработной платы.
  - У каждого работника должны иметься служебные обязанности, поэтому отношение является обязательным.
  - Работник может занимать разные должности в течение срока пребывания в компании. Таким образом, у работника может иметься большое количество должностных обязанностей. Сведения о них могут быть сохранены как история, если произойдет смена должности работника, либо может получиться так, что работник одновременно занимает две разные должности. Например, начальник отдела может принять на себя обязанности работника, если тот уволится, а человек ему на замену еще не будет нанят.
- `Spouse`.
  - В этом упрощенном примере класс `Spouse` содержит только дату годовщины.
  - Работник может состоять или не состоять в браке. Таким образом, `Spouse` является необязательным.
  - У работника может быть только один супруг.
- `Child`.
  - В этом упрощенном примере класс `Child` содержит только строку `FavoriteToy`.
  - У работника могут иметься или не иметься дети.
  - У работника может не быть детей либо иметься несметное количество детей (ого!). Вы могли бы принять проектное решение относительно верхнего предела количества детей, с которым сможет справиться система.

Чтобы можно было подытожить все это, в табл. 9.1 представлена кардинальность ассоциаций классов, которые мы только что рассмотрели.

Таблица 9.1. Кардинальность ассоциаций классов

Необязательная/Ассоциация	Кардинальность	Обязательная
Employee/Division	1	Обязательная
Employee/JobDescription	1...n	Обязательная
Employee/Spouse	0...1	Необязательная
Employee/Child	0...n	Необязательная

НОТАЦИЯ КАРДИНАЛЬНОСТИ

Нотация 0...1 означает, что у работника может не быть супруга либо иметься один супруг. Нотация 0...n означает, что у работника может быть любое количество детей от нуля до бесконечности.

На рис. 9.7 показана диаграмма класса для рассматриваемой нами системы. Обратите внимание, что на этой диаграмме класса кардинальность указана вдоль ассоциативных линий. Загляните в табл. 9.1, чтобы узнать, является ли определенная ассоциация обязательной.

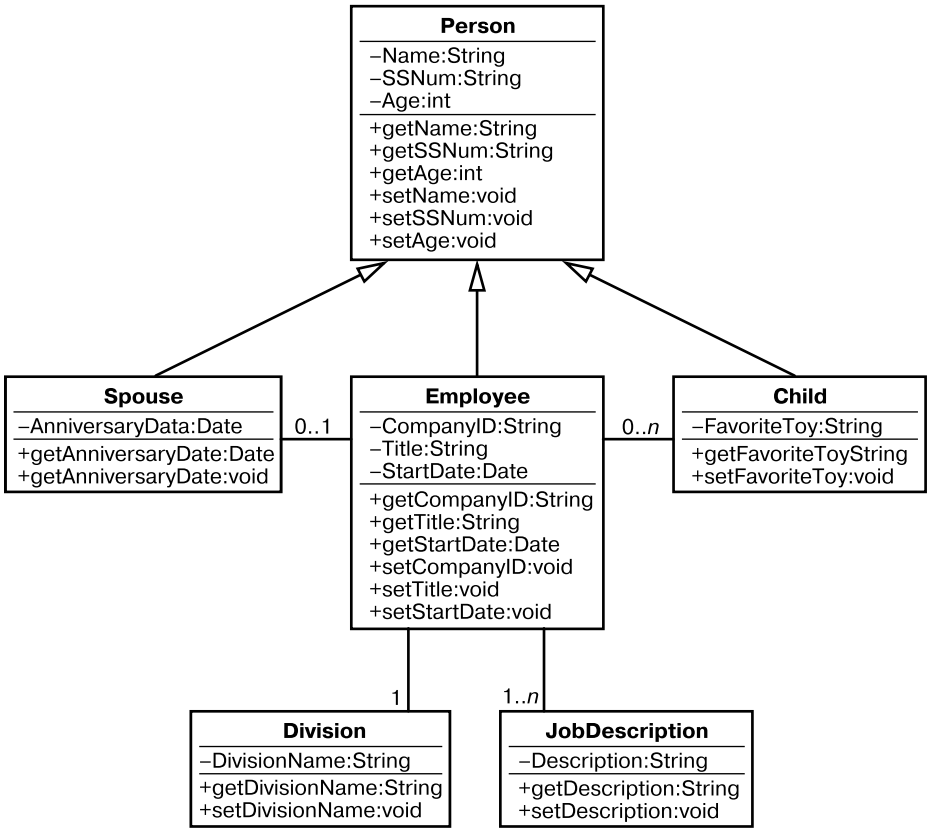

Рис. 9.7. Кардинальность на UML-диаграмме

Ассоциации, включающие множественные объекты

Как нам представить ассоциацию, которая может включать множественные объекты (например, от 0 до большого количества детей) в коде? Вот код для класса `Employee`:

```
import java.util.Date;

public class Employee extends Person{

    private String CompanyID;
    private String Title;
```

```
    private Date StartDate;

    private Spouse spouse;
    private Child[] child;
    private Division division;
    private JobDescription[] jobDescriptions;

    public String getCompanyID() {return CompanyID;}
    public String getTitle() {return Title;}
    public Date getStartDate() {return StartDate;}

    public void setCompanyID(String CompanyID) {}
    public void setTitle(String Title) {}
    public void setStartDate(int StartDate) {}

}
```

Обратите внимание, что классы, для которых имеет место отношение «один-комногим», представлены в коде массивами:

```
private Child[] child;
private JobDescription[] jobDescriptions;
```

Необязательные ассоциации

Когда речь идет об ассоциациях, одним из наиболее важных аспектов является необходимость позаботиться о проектировании вашего приложения таким образом, чтобы оно выполняло проверку необязательных ассоциаций. Это означает, что ваш код должен проверять, имеет ли место `null` для определенной ассоциации.

Допустим, в приводившемся ранее примере ваш код полагает, что у каждого работника есть супруг. Однако если один из работников окажется не состоящим в браке, то у кода возникнет проблема (рис. 9.8). Если ваш код в действительности ожидает, что супруг будет иметься, то при его выполнении вполне может произойти сбой, что ввергнет систему в нестабильное состояние. Важно, чтобы код проводил проверку на предмет условия `null`, а также обрабатывал его как допустимое условие.

Например, если супруг отсутствует, то код не должен пытаться вызывать метод `Spouse`, иначе это может привести к сбою приложения. Таким образом, код должен быть способен обработать объект `Employee`, у которого нет `Spouse`.

Связываем все воедино: пример

Поработаем над простым примером, который свяжет воедино концепции наследования, интерфейсов, композиции, ассоциаций и агрегаций в рамках одной компактной системной диаграммы.

Снова обратимся к примеру, приводившемуся в главе 8, но на этот раз с одним дополнением: мы добавим класс `Owner`, который будет содержать поведение `walkDog`.

Рис. 9.8. Проверка всех необязательных ассоциаций

Напомню, что класс `Dog` наследует напрямую от класса `Mammal`. Сплошная стрелка представляет это отношение между классами `Dog` и `Mammal` на рис. 9.9. Класс `Nameable` является интерфейсом, реализуемым `Dog`, что демонстрирует штриховая стрелка от класса `Dog` к интерфейсу `Nameable`.

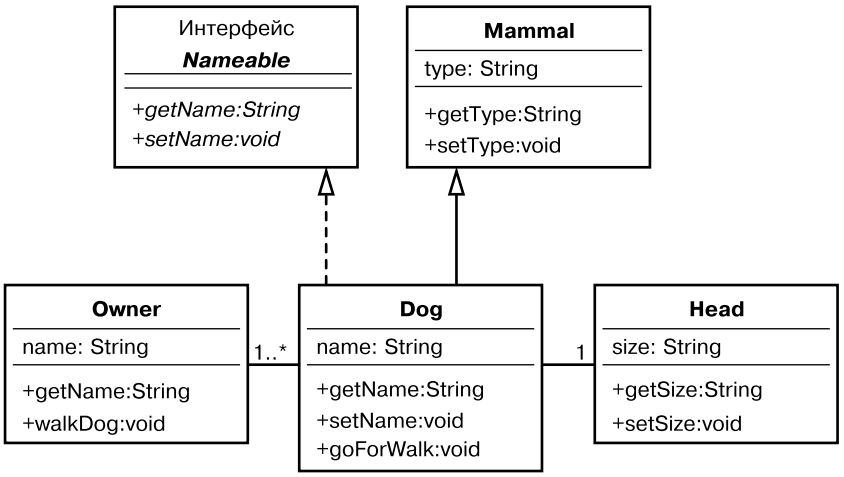

Рис. 9.9. UML-диаграмма для примера с Dog

В этой главе мы главным образом рассматривали ассоциации и агрегации. Отношение между классами `Dog` и `Head` считается отношением агрегации, поскольку `Head` на самом деле является частью `Dog`. Кардинальность, указанная над линией,

соединяющей эти два класса на диаграмме, определяет, что для Dog может иметься только один класс Head.

Отношение между классами Dog и Owner является отношением ассоциации. Owner, разумеется, не является частью Dog и наоборот, поэтому мы можем с уверенностью исключить отношение агрегации. Однако классу Dog требуется услуга от Owner — поведение walkDog. Кардинальность, указанная над линией, соединяющей классы Dog и Owner, определяет, что для Dog может иметься один или более классов, представляющих владельцев (например, как жену, так и мужа можно считать владельцами, совместно отвечающими за то, чтобы выводить собаку на прогулку).

Определение этих отношений наследования, интерфейса, композиции, ассоциации и агрегации станет основной частью проектной работы, с которой вы будете сталкиваться при проектировании объектно-ориентированных систем.

ГДЕ HEAD?

Вы, возможно, решите, что имеет смысл присоединить класс Head к классу Mammal, а не к классу Dog, поскольку в реальной жизни у всех млекопитающих вроде бы имеется голова. В этой модели я использовал класс Dog как центральный элемент примера, в силу чего и прикрепил Head к Dog.

Резюме

В текущей главе мы разобрали некоторые особенности композиции, а также два ее типа — агрегацию и ассоциацию. В то время как наследование представляет новую разновидность уже существующего объекта, композиция представляет взаимодействия между разными объектами.

В трех последних главах мы рассмотрели основы наследования и композиции. Используя эти концепции и применяя свои навыки в процессе разработки программного обеспечения, вы сможете проектировать качественные классы и объектные модели. В следующей главе мы поговорим о том, как использовать UML-диаграммы классов в качестве средств, помогающих создавать объектные модели.

Ссылки

- *Буч Гради, Максимчук Роберт А., Энгл Майкл У., Янг Бобби Дж., Коналлен Джим и Хьюстон Келли А.* Объектно-ориентированный анализ и проектирование с примерами приложений (Object-Oriented Analysis and Design with Applications). 3-е изд. — Бостон: Addison-Wesley, 2007.
- *Майерс Скотт.* Эффективное использование C++ (Effective C++). 3-е изд. — Бостон: Addison-Wesley Professional, 2005.
- *Коуд Петер и Мейфилд Марк.* Проектирование на Java (Java Design). — Апер-Сэддл-Ривер: Prentice-Hall, 1997.
- *Гилберт Стивен и Маккарти Билл.* Объектно-ориентированное проектирование на Java (Object-Oriented Design in Java). — Беркли: The Waite Group Press (Pearson Education), 1998.

Глава 10
Создание объектных моделей

Я глубоко убежден в том, что прежде чем осваивать конкретные средства моделирования, необходимо изучить фундаментальные объектно-ориентированные концепции. Поэтому выбор того, где именно поместить эту главу, был несколько проблематичным. Во многих отношениях эта глава могла бы идти первой, поскольку UML-диаграммы представлены на всем протяжении этой книги, включая главу 1. В конце концов я решил поместить эту главу после «концептуальных» глав, которыми считаю главы 1–9. В оставшихся главах рассматриваются специфические вопросы, касающиеся программных технологий.

Эта глава представляет собой краткий обзор нотации UML-диаграмм классов, использованной в книге, которую вы сейчас читаете. Однако ее нельзя назвать полным руководством по UML, поскольку для рассмотрения только одной этой темы потребовалась бы целая книга, которых и так написано большое количество. Хорошие источники информации приводятся в конце текущей главы. Поскольку в этой книге рассматриваются основы, в ней лишь поверхностно затрагивается то, что способен предложить язык UML.

Используемые нами в этой книге UML-диаграммы классов касаются моделирования объектно-ориентированных систем, или, как я люблю называть его, *объектного моделирования*. Соответствующая нотация задействует диаграммы классов в целях системного моделирования. В данной книге не рассматриваются многие компоненты UML, например диаграммы состояний и диаграммы активности. Каждая из этих тем могла бы заслуживать целой главы или более. Опять-таки назначение текущей главы заключается в обеспечении краткого обзора объектных моделей и в особенности диаграмм классов, так что если вы не знакомы с диаграммами классов, то сможете быстро овладеть соответствующими основами. Благодаря такому введению примеры в этой книге будут более выразительными.

Что такое UML

UML, как видно из его названия, является языком моделирования. В книге «Язык UML. Руководство пользователя» *(The UML User Guide)* UML определяется как «графический язык для визуализации, специфицирования, конструирования и документирования артефактов систем, в которых главная роль принадлежит программному обеспечению». Язык UML обеспечивает стандартный способ создания проектов систем. Коротко говоря, UML позволяет графически представлять объектно-ориентированные программные системы и манипулировать ими. Он не только дает возможность создавать представления конструкций систем, но и является инструментом, который помогает их проектировать.

UML — это синтез разных языков моделирования, независимо разработанных Гради Бучем (Grady Booch), Джеймсом Рамбо (James Rumbaugh) и Иваром Якобсоном (Ivar Jacobson), которые все вместе получили ласковое прозвище «Три амиго». Компания — разработчик программного обеспечения Rational Software свела вместе три языка моделирования под одной крышей — таким образом, то, что получилось в итоге, стало называться унифицированным языком моделирования. Как уже отмечалось ранее, объектное моделирование представляет собой одну из частей языка UML.

Однако важно не слишком тесно связывать язык UML с объектно-ориентированной разработкой. В своей статье под названием «Что такое UML и чем он не является» *(What the UML Is — and Isn't)* Крейг Ларман пишет:

> «К сожалению, в контексте разработки программного обеспечения и языка UML, позволяющего создавать диаграммы, умение читать и писать UML-нотацию, похоже, иногда приравнивается к навыкам в объектно-ориентированном анализе и проектировании. Конечно, на самом деле это не так, и последнее из упомянутого намного важнее первого. Поэтому я рекомендую искать учебные курсы и материалы, в которых приобретению интеллектуальных навыков в объектно-ориентированном анализе и проектировании придается первостепенное значение по сравнению с UML-нотацией или использованием средств автоматизированной разработки программного обеспечения».

Несмотря на то что знание языка UML очень важно, намного важнее сначала приобрести объектно-ориентированные навыки. Изучение UML до освоения объектно-ориентрованных концепций аналогично попытке научиться читать электрические схемы, изначально ничего не зная об электричестве.

Структура диаграммы класса

Диаграмма класса состоит из трех частей: имени класса, его атрибутов и методов (конструкторов, считающихся методами). Диаграмма класса по сути является прямоугольником, в котором эти три части разделены горизонтальными линиями. Вернемся к примеру с таксистом. На рис. 10.1 показана UML-диаграмма, представляющая класс `Cabbie`.

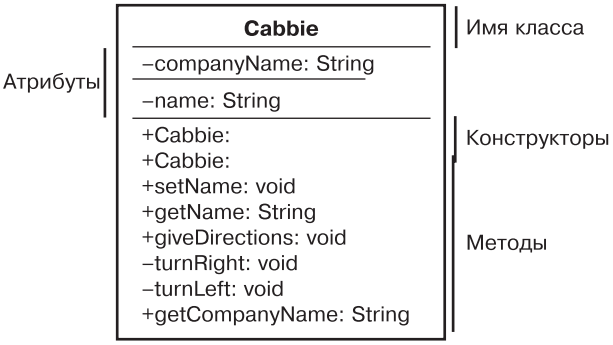

Рис. 10.1. UML-диаграмма класса Cabbie

Эта UML-диаграмма в точности соответствует приведенному далее Java-коду:

```
/*

  Этот класс определяет Cabbie и присваивает Cab
*/
public class Cabbie {

  // Место для указания значения companyName
  private static String companyName = "Blue Cab Company";

  // Атрибут name, относящийся к Cabbie
  private String name;

  // Cab, присвоенный Cabbie

  // Конструктор по умолчанию для Cabbie
  public Cabbie() {

    name = null;
    myCab = null;

  }

  // Инициализация конструктора для Cabbie
  public Cabbie(String iName, String serialNumber) {

    Name = iName;
    myCab = new Cab(serialNumber);

  }

  // Задание значения для name, относящегося к Cabbie
  public void setName(String iName) {
    name = iName;
  }

  // Извлечение значения для name, относящегося к Cabbie
  public String getName() {
    return name;
  }

  // Выполнение метода giveDirections класса Cabbie
  public void giveDirections(){
  }

  // Выполнение метода turnRight класса Cabbie
  private void turnRight(){
  }

  // Выполнение метода turnLeft класса Cabbie
  private void turnLeft() {
```

```
    }

    // Извлечение значения companyName
    public static String getCompanyName() {
      return companyName;
    }

  }
```

Уделите минуту тому, чтобы взглянуть на этот код и сравнить его с соответствующей UML-диаграммой класса. Обратите внимание на то, как имя класса, его атрибуты и методы в коде соотносятся с обозначениями на диаграмме класса. Собственно, вот и все, что касается диаграммы класса в плане структуры. Однако из диаграммы можно почерпнуть намного больше информации. Об этой информации мы и поговорим в следующих разделах.

Атрибуты и методы

Помимо представления структуры класса его диаграмма также дает информацию о соответствующих атрибутах и методах.

Атрибуты

Обычно об атрибутах не думают как об имеющих подписи, которые есть у методов. Однако тот или иной атрибут имеет тип, обозначаемый на диаграмме класса. Взгляните на два атрибута из примера с `Cabbie`:

```
-companyName:String
-name:String
```

Оба атрибута определены как `String`. Здесь все представлено следующим образом: сначала идет имя атрибута, а затем — его тип (в обоих случаях это `String`). Там также могли быть атрибуты, определенные как `int` и `float`, как в этом примере:

```
-companyNumber:float
-companyAge:int
```

Взглянув на диаграмму класса, вы сможете сказать, какой тип данных у определенного параметра. Вы также сможете сказать, какие атрибуты объявлены как `private` благодаря знаку минуса (–), стоящему перед ними. Знак плюса (+) означал бы, что они объявлены как `public`. Исходя из всего рассмотренного нами в предыдущих главах вы знаете, что все атрибуты следует объявлять как `private`. Время от времени некоторые люди приводят доводы в пользу применения методов, объявленных как `public`, однако описываемый в этой книге подход предполагает объявление атрибутов как `private` во всех ситуациях.

Методы

Та же логика, которая имеет место, когда речь идет об атрибутах, работает и в случае с методами. Вместо того чтобы выражать тип, диаграмма показывает возвращаемый тип соответствующего метода.

Если вы взглянете на приведенный далее фрагмент из примера с Cabbie, то увидите, что там представлено имя метода наряду с возвращаемым типом и модификатором доступа (например, public, private):

```
+Cabbie:
+giveDirections:void
+getCompanyName:String
```

Во всех трех случаях присутствует модификатор доступа public (что обозначается знаком плюса). Если бы тот или иной метод был объявлен как private, то перед ним стоял бы знак минуса. После имени каждого из методов идет двоеточие, отделяющее имя метода от возвращаемого типа.

Можно включить список параметров, сделав это следующим образом:

```
+getCompanyName(список параметров):String
```

Параметры в списке параметров разделяются запятыми:

```
+getCompanyName(параметр1, параметр2, параметр3):String
```

Я предпочитаю делать так, чтобы объектные модели были как можно проще, поэтому обычно включаю в диаграммы классов только имя соответствующего класса, его атрибуты и методы. Это позволяет сконцентрировать внимание на конструкции в целом, не сосредотачиваясь на деталях. Включение в диаграммы классов слишком большого количества информации (например, параметров) делает объектную модель трудной для восприятия.

Обозначения доступа

Как отмечалось ранее, знаки плюса (+) и минуса (–), располагающиеся слева от атрибутов и методов, указывают, являются эти атрибуты и методы открытыми или закрытыми. Атрибут или метод считается закрытым, если перед ним стоит знак минуса. Это означает, что никакой другой класс не сможет получить доступ к этому атрибуту или методу; только у методов в соответствующем классе есть возможность инспектировать или изменять его.

Если слева от атрибута или метода стоит знак плюса, то этот атрибут или метод является открытым и любой класс сможет инспектировать или модифицировать его. Взгляните, к примеру, на следующий код:

```
-companyNumber:float
+companyAge:int
```

В этом примере companyNumber является закрытым, и только методы его класса могут что-либо делать с ним. Однако companyAge является открытым, в силу чего любому классу разрешен доступ к нему и дана возможность модифицировать его.

Если в коде нет никаких обозначений доступа, то система считает, что в данном случае предусмотрен доступ по умолчанию, и при этом не используется знак плюса или минуса:

```
companyNumber:float
companyAge:int
```

В Java типом доступа по умолчанию является `protected`. Он означает, что только классы в соответствующем пакете могут получить доступ к определенному атрибуту или методу. Java-пакет — это набор связанных классов, которые были намеренно сгруппированы разработчиком.

Согласно информации из MSDN-сети компании Microsoft, в .NET есть следующие модификаторы доступа:

- `public` — доступ к типу или члену возможен из любого другого кода в той же сборке или иной сборке, которая на него ссылается;
- `private` — доступ к типу или члену возможен только из кода в том же классе или структуре;
- `protected` — доступ к типу или члену возможен только из кода в том же классе или структуре или производном классе;
- `internal` — доступ к типу или члену возможен из любого кода в той же сборке, но не из кода в другой сборке. В Objective-C тоже предусмотрены модификаторы открытого (`@public`) и закрытого доступа (`@private`). По умолчанию в Objective-C предусмотрен защищенный доступ, то есть возможен доступ к членам соответствующего подкласса.

Наследование

Чтобы понять, как представляется наследование, взгляните на пример с `Dog` из главы 7. В этом примере класс `GoldenRetriever` наследует от класса `Dog`, как показано на рис. 10.2. Это отношение представлено на UML стрелкой, указывающей в направлении родительского класса, или суперкласса.

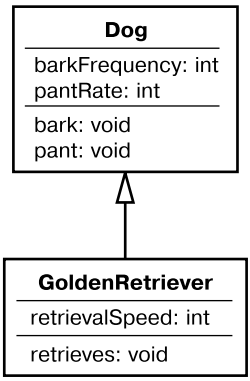

Рис. 10.2. UML-диаграмма иерархии во главе с Dog

Нотация в данном случае проста, при этом стрелка используется как индикатор отношения наследования.

ИНДИКАЦИЯ НАСЛЕДОВАНИЯ ИНТЕРФЕЙСОВ

Штриховые стрелки — это индикаторы интерфейсов, о которых пойдет речь в следующем разделе.

Поскольку примеры кода в этой книге написаны главным образом на Java, нам не нужно беспокоиться о множественном наследовании. Аналогичным образом дело обстоит и в случае с .NET и Objective-C. Язык C++ реализует множественное наследование.

Однако несколько подклассов может наследовать от одного и того же суперкласса. Опять-таки мы можем воспользоваться примером с `Dog` из главы 7 (рис. 10.3).

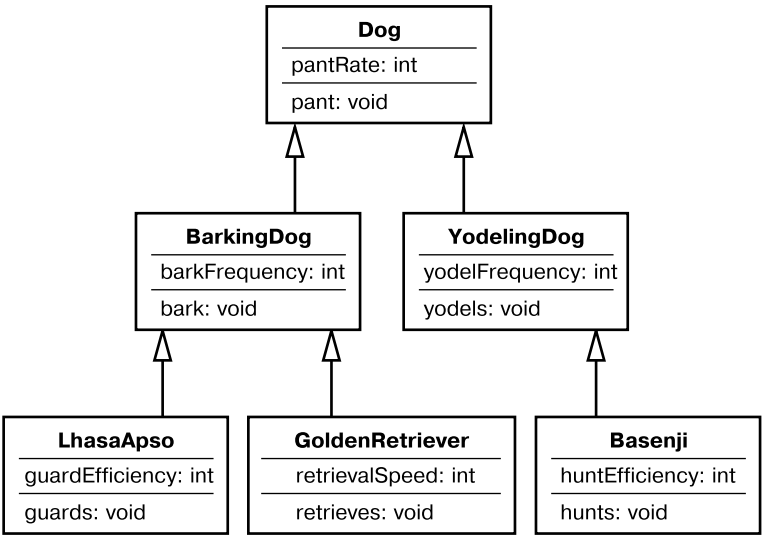

Рис. 10.3. UML-диаграмма расширенной иерархии во главе с Dog

Этот пример иллюстрирует две концепции при моделировании дерева наследования. Первая заключается в том, что у суперкласса может иметься более одного подкласса. Вторая состоит в том, что дерево наследования может простираться более чем на один уровень. В примере на рис. 10.3 показаны три уровня. Мы могли бы добавить новые уровни, включив специфические разновидности ретриверов или даже внедрив более высокий уровень путем создания класса `Canine` (рис. 10.4).

МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ

Если вы проектируете с использованием таких языков программирования, как Eiffel и C++, то знайте, что в них встроено множественное наследование, которое будет частью общей конструкции.

Интерфейсы

Поскольку интерфейсы — это особый тип наследования, нотации схожи и могут вызывать замешательство. Ранее отмечалось, что стрелки показывают наследование. Интерфейсы тоже обозначаются стрелками, но только штриховыми. Такая нотация является индикатором отношения между наследованием и интерфейсами, однако также разграничивает их. Взгляните на рис. 10.5, на котором показана

сокращенная версия примера из главы 8. Класс `Dog` наследует от класса `Mammal` и реализует интерфейс `Nameable`.

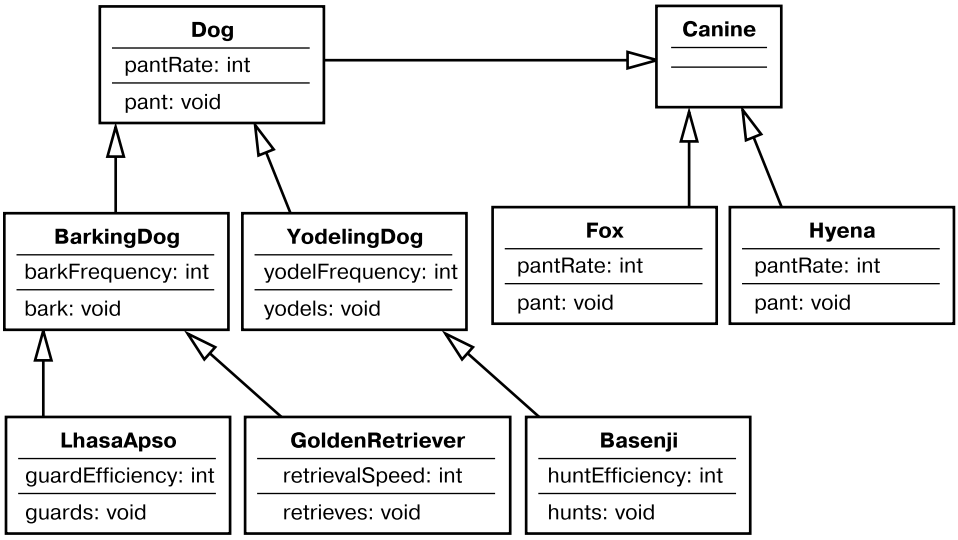

Рис. 10.4. UML-диаграмма иерархии, включающей класс Canine

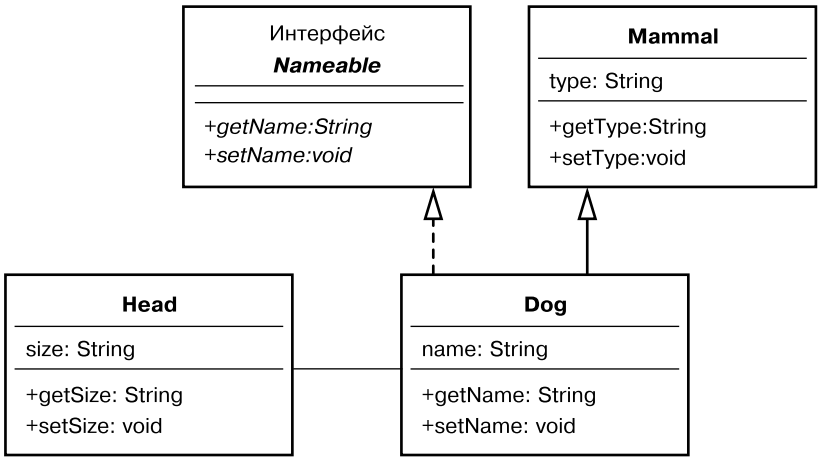

Рис. 10.5. UML-диаграмма отношения интерфейса

Композиция

Композиция — индикатор использования отношения «содержит как часть». Когда наследование не оказывается правильным выбором при проектировании (из-за того, что отношение «является экземпляром» не подходит), обычно используется композиция.

В главе 9 мы рассмотрели два типа композиции — агрегацию и ассоциацию. Композиция используется при создании классов с применением других классов. При агрегации один класс является компонентом другого класса (как, например, покрышка по отношению к автомобилю). При ассоциации одному классу требуются услуги другого класса (например, когда клиенту требуются услуги сервера).

Агрегации

Агрегация представляется линией, на конце которой имеется ромб. Чтобы показать, что в примере с классом `Car` из главы 9 объект `SteeringWheel` является частью этого класса, была использована нотация, которая демонстрируется на рис. 10.6.

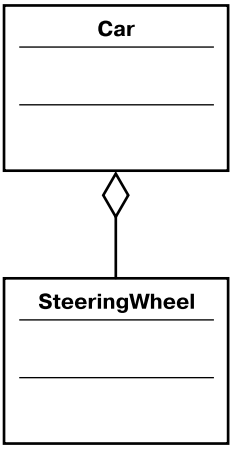

Рис. 10.6. UML-диаграмма, представляющая композицию

Как и в случае с деревом наследования, количество возможных уровней агрегации неограниченно (теоретически). В примере, показанном на рис. 10.7, есть четыре уровня. Следует отметить, что разные уровни могут представлять разные агрегации. Например, `Stereo` является частью `Car`, `Radio` — частью `Stereo`, а `Tuner` — частью `Radio`.

Ассоциации

В то время как агрегации представляют части целого, подразумевая, что один класс логически строится из частей другого класса, ассоциации — это услуги, которые классы оказывают друг другу.

Как уже отмечалось ранее, отношение «клиент/сервер» вписывается в эту модель. Хотя ясно, что клиент не является частью сервера, точно так же, как сервер не является частью клиента, они оба зависят друг от друга. В большинстве случаев вы можете сказать, что сервер предоставляет клиенту услугу. В UML-нотации эту услугу обозначает простая линия, на обоих концах которой нет никаких фигур (рис. 10.8).

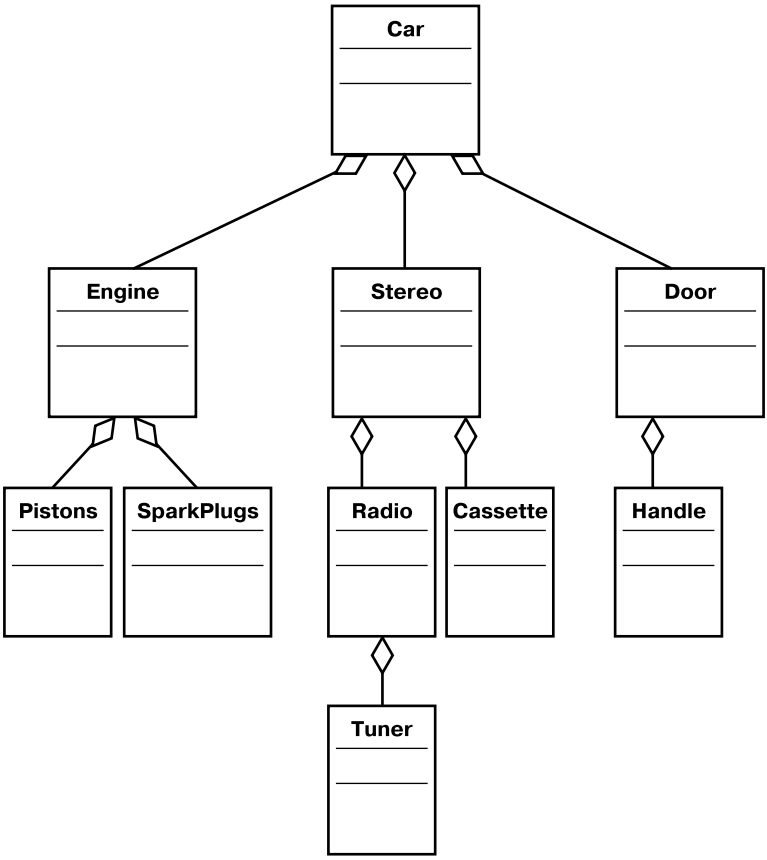

Рис. 10.7. Расширенная UML-диаграмма, представляющая композицию

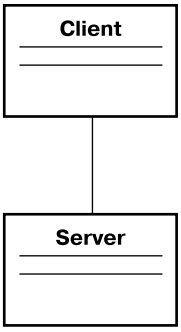

Рис. 10.8. UML-диаграмма, представляющая ассоциацию

Обратите внимание, что, поскольку ни на одном из концов линии нет каких-либо фигур, индикатор направления потока услуги отсутствует. На рисунке лишь показано, что между двумя классами имеется ассоциация.

В качестве иллюстрации взгляните на пример компьютерной системы, приведенный в главе 9. В данном случае имеется несколько компонентов: `Computer`, `Monitor`, `Scanner`, `Keyboard` и `Mouse`. Каждый из них представляет собой полностью отдельный компонент, в некоторой степени взаимодействующий с `Computer` (рис. 10.9).

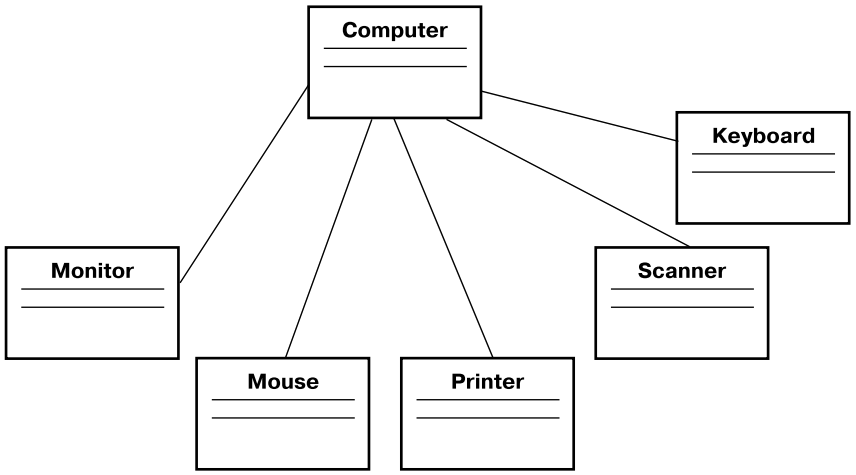

Рис. 10.9. Расширенная UML-диаграмма, представляющая ассоциацию

Здесь важно отметить, что `Monitor` технически является частью `Computer`. Если бы вам потребовалось создать класс для компьютерной системы, то вы могли бы смоделировать его, использовав агрегацию. Однако `Computer` представляет собой определенную форму агрегации, поскольку состоит из `Motherboard`, `RAM` и т. д. (рис. 10.10).

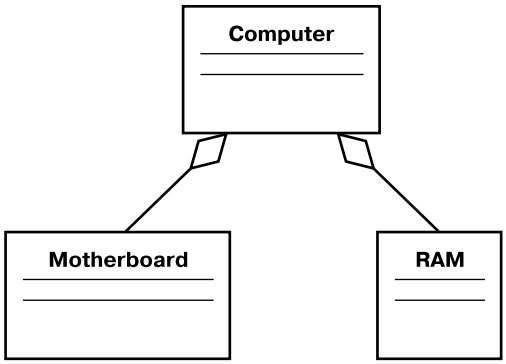

Рис. 10.10. UML-диаграмма, представляющая агрегацию

Кардинальность

В этой главе осталось рассмотреть только кардинальность. По сути она связана с диапазоном объектов, соответствующих определенному классу. Используя

приводившийся ранее пример компьютерной системы, мы можем сказать, что в состав компьютера входит одна и только одна материнская плата. Компьютер никак не может обойтись без материнской платы, а в настоящее время среди персональных компьютеров нет таких, в которых было бы несколько материнских плат. С другой стороны, в компьютере должна быть установлена как минимум одна планка оперативной памяти, однако в нем может быть установлено столько планок оперативной памяти, сколько поддерживается его материнской платой. Таким образом, мы можем представить кардинальность как `1...n`, где `n` является бесконечным числом — по крайней мере, в общем смысле.

ПРЕДЕЛЬНЫЕ ЗНАЧЕНИЯ КАРДИНАЛЬНОСТИ

Если нам известно, что для установки планок оперативной памяти предусмотрено шесть слотов, то число, определяющее верхний предел, не будет бесконечным. Таким образом, вместо n было бы 6, а кардинальность обозначалась бы как 1...6.

Взгляните на пример, приведенный на рис. 9.7. В этом примере продемонстрировано несколько представлений кардинальности. Во-первых, у класса `Employee` имеется ассоциация с классом `Spouse`. У работника может не быть супруга либо иметься один супруг (по крайней мере, в нашей культуре у работника не может быть более одного супруга). Таким образом, кардинальность этой ассоциации представлена как `0...1`.

Ассоциация между классом `Employee` и классом `Child` несколько отличается тем, что работник теоретически не ограничен в количестве детей, которое у него может быть. У работника может совсем не быть детей, но если они у него все же имеются, то верхнего предела их количества не существует. Таким образом, кардинальность этой ассоциации представлена как `0...n`, а `n` при этом означает отсутствие верхнего предела количества детей, с которым сможет справиться система.

Отношение между `Employee` и классом `Division` говорит о том, что каждый работник может быть ассоциирован с одним и только одним отделом. Это отношение представляет простая единица. Индикатор кардинальности выступает как очень важная часть объектной модели.

ДОПОЛНИТЕЛЬНЫЕ ЗАДАЧИ ПРОЕКТИРОВАНИЯ

При определенных обстоятельствах работник может быть ассоциирован с несколькими отделами. Например, в колледже один человек может одновременно занимать две должности на факультетах математики и информатики. Это еще одна задача проектирования, которую вы должны принимать во внимание.

Последней мы рассмотрим ассоциацию кардинальности между классами `Employee` и `JobDescription`. В этой системе у работника может быть неограниченное количество должностных обязанностей. Однако в отличие от класса `Child`, где количество детей может быть нулевым, в этой системе на каждого работника должна приходиться по крайней мере одна должностная обязанность. Таким образом, кардинальность этой ассоциации представлена как `1...n`. Эта ассоциация подразумевает как минимум одну должностную обязанность на каждого работника, однако их может быть и больше (в данном случае неограниченное количество).

СОХРАНЕНИЕ ИСТОРИИ

Вы также должны принимать во внимание то, что у работника могут оставаться обязанности, связанные с прошлой должностью. При этом необходим способ разграничения текущих и прошлых должностных обязанностей. Он может быть обеспечен благодаря наследованию путем создания набора объектов, представляющих разные должности, с атрибутом, который будет индикатором того, какая должность является текущей.

Резюме

В данной главе приведен очень краткий обзор нотации UML-диаграмм классов, использованной в этой книге. Как уже отмечалось во введении, язык UML — очень сложная и важная тема, а полное ее рассмотрение само по себе займет целую книгу (или несколько книг).

Диаграммы классов применяются для иллюстрирования объектно-ориентированных примеров на всем протяжении этой книги. Вам не потребуется язык UML для проектирования объектно-ориентированных систем, однако он может стать подходящим инструментом при разработке таких систем. Вместе с тем мне очень нравится использовать диаграммы классов в качестве визуального средства при создании объектных моделей.

Детальное изучение языка UML — одно из действий, которые вам следует предпринять после того, как вы усвоите основополагающие объектно-ориентированные концепции. Однако, как это часто бывает, возникает вопрос: «Что было раньше — курица или яйцо?» Язык UML оказался очень полезным при иллюстрации некоторых примеров, приведенных в этой книге.

При объяснении объектно-ориентированных концепций целесообразно применять и язык моделирования (например, UML), и язык программирования (например, Java). Разумеется, мы могли бы воспользоваться C++ вместо Java, а также другим языком моделирования взамен UML. Важно помнить, что независимо от того, какие примеры вы разбираете, вам следует оставаться сосредоточенными на объектно-ориентированных концепциях как таковых.

К настоящему времени в этой книге было рассмотрено много материала, относящегося к объектно-ориентированным концепциям. Я старался обеспечить высокоуровневый обзор концепций, имеющих место при объектно-ориентированном мышлении. Были подробно рассмотрены инкапсуляция, наследование, полиморфизм и композиция. В оставшейся части этой книги мы сосредоточимся на некоторых объектно-ориентированных технологиях.

Ссылки

- *Шмуллер Джозеф*. Освой самостоятельно UML за 24 часа (Teach Yourself UML in 24 Hours). 3-е изд. — Индианаполис: Sams Publishing, 2006.
- *Буч Г., Якобсон И. и Рамбо Д.* Язык UML. Руководство пользователя (The UML User Guide). 2-е изд. — Бостон: Addison-Wesley, 2005.

- *Ли Ричард и Тепфенхарт Уильям*. Практическая объектно-ориентированная разработка с использованием UML и Java (Practical Object-Oriented Development with UML and Java). — Аппер-Сэддл-Ривер: Prentice-Hall, 2003.
- *Амблер Скотт*. Элементы UML-стиля (The Elements of UML Style). — Кембридж: Cambridge University Press, 2003.
- *Фаулер Мартин*. UML. Основы (UML Distilled). 3-е изд. — Бостон: Addison-Wesley Professional, 2003.
- *Ларман Крейг*. Что такое UML и чем он не является (What the UML Is — and Isn't) / Java Report. — 1999. — № 4(5). — С. 20–24.

Глава 11

Объекты и переносимые данные: XML и JSON

За последнее десятилетие объектно-ориентированные технологии получили значительное распространение. Объекты стали одной из основных технологий в индустрии разработки приложений. Кроме того, объекты существенно прогрессировали в плане определения и перемещения данных. На протяжении нескольких последних лет вокруг переносимости кода царит большой ажиотаж. Например, успех языка программирования Java в значительной мере обусловлен тем, что написанный на нем код очень легко переносить между разными платформами.

Байт-коды на Java могут выполняться на разных платформах, если в системе загружена виртуальная машина Java. Фреймворк .NET обеспечивает переносимость другого очень важного типа — переносимость между разными языками программирования. Сборки, созданные с применением C# .NET, можно использовать в приложениях, написанных на Visual Basic .NET или, собственно говоря, любом другом .NET-языке программирования. Конечно, несмотря на принятие множества технологических стандартов, основные языки программирования (Java, .NET, Objective-C и др.) по-прежнему остаются проприетарными. Возможно, в будущем появится язык программирования, позволяющий писать код, который будет полностью и экономически переносимым как между разными языками программирования, так и между платформами.

Хотя языки программирования, позволяющие писать переносимый код, являются мощными инструментами, в действительности они составляют лишь половину «уравнения» при разработке приложений. Программы, написанные на этих языках, должны обрабатывать данные, которые должны быть превращены в информацию. Эта информация управляет бизнесом. Информация — это и есть вторая половина «уравнения» портативности.

XML — это стандартный механизм определения и переноса данных между потенциально несопоставимыми системами (JSON — еще один доступный вариант). При использовании объектно-ориентированных языков программирования, например Java, VB и C#, в сочетании с объектно-ориентированными языками определения данных, например XML и JSON, перемещение данных между разными локациями оказывается намного более эффективным и безопасным. XML и JSON обеспечивают механизм, позволяющий независимым приложениям обмениваться данными.

Переносимые данные

Исторически сложилось так, что многообразие форматов хранения данных представляет собой большую проблему для бизнеса. Допустим, компания «Альфа» использует систему баз данных Oracle для управления своей системой сбыта. Предположим также, что компания «Бета» применяет систему баз данных SQL Server для управления своей системой закупок. Теперь рассмотрим проблему, которая возникнет, когда компании «Альфа» и «Бета» захотят заниматься бизнесом через Интернет. Хотя при создании системы надлежит решить множество вопросов, здесь мы займемся проблемой, состоящей в том, что две проприетарные базы данных не являются напрямую совместимыми. Наша цель — создать такую электронную систему заказов на покупку для компании «Бета», использующей SQL Server, которая будет напрямую взаимодействовать с системой сбыта компании «Альфа», использующей систему баз данных Oracle.

Кроме того, многим компаниям требуется передавать информацию внутри своих организаций, а также в другие компании. Значительная часть электронной торговли ведется через Интернет и локальные интрасети, а типы бизнес-систем, которые подразумевают электронную торговлю, довольно разнообразны.

Язык XML предусматривает стандарты для перемещения данных разными способами. Данные зачастую можно представлять себе как перемещающиеся вертикально и горизонтально. Термин *«вертикально»* означает, что данные предназначены для перемещения через множественные отраслевые группы. Отраслевые группы, например те, что существуют в сфере учета и финансов (где используется, например, FpML — язык разметки финансовых продуктов), разработали собственные языки разметки, предполагающие стандартные определения данных. Соответствующие вертикальные приложения предусматривают специфические бизнес-модели и терминологию относительно перемещения информации через разные отрасли. Совокупность этих стандартов часто называют *словарем*. Таким образом, отраслевые группы используют XML для формирования словаря.

Другой подход к XML-стандартам наблюдается в случае с *горизонтальными* приложениями. Такие приложения специфичны для определенной отрасли вроде розничной торговли или перевозок. Во всех приложениях для электронной коммерции первостепенное значение имеет обмен данными. На рис. 11.1 показано, как данные могут перемещаться вертикально и горизонтально через разные отрасли.

Один из интересных образцов отраслевого применения XML — пример языка разметки рецептов Recipe Markup Language (RecipeML). RecipeML — это XML-словарь, определяющий стандарты для связанных с питанием отраслей вроде гостиничного и ресторанного дела и т. д. Использование RecipeML делает данные переносимыми и позволяет отраслям перемещать их туда-сюда стандартным образом. К числу отраслей, в которых используются стандарты на основе XML, относятся юриспруденция, учет, розничная торговля, туризм, финансы и образование. При работе в этих областях нужно рассматривать концепцию переносимых данных. Хотя низкоуровневые данные (на машинном уровне), конечно же, не являются переносимыми, мы хотим обеспечить переносимость более высокого уровня (на информационном уровне). В то время как Java, .NET и Objective-C обеспечивают

разные степени переносимости на уровне языка программирования, XML предоставляет требуемую нам переносимость информации.

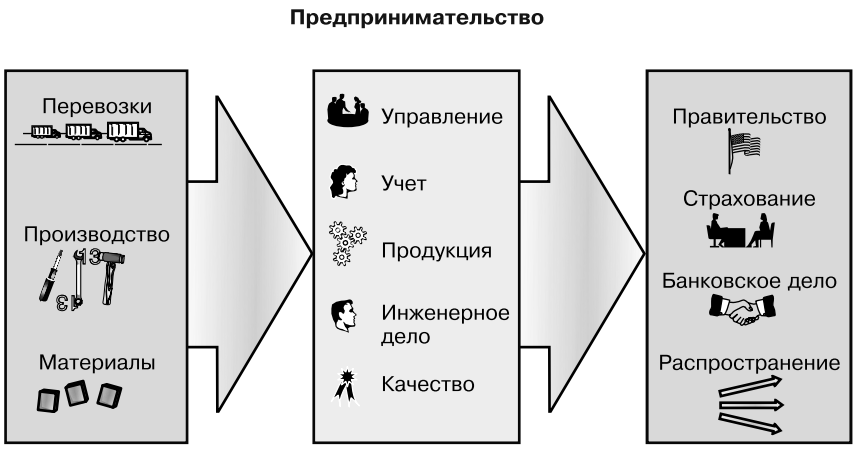

Рис. 11.1. Перемещение данных с использованием XML через разные отрасли

XML

XML означает «расширяемый язык разметки». Вам, вероятно, уже знаком язык разметки гипертекста HTML. Оба языка являются потомками SGML — стандартного обобщенного языка разметки. Удивительно то, что SGML появился еще в 1970-х годах, а стандартизирован был в 1980-х годах.

Основная функция языка HTML — представлять данные в браузерах. Этот язык был разработан для организации данных с использованием гиперссылок, а браузеры идеально подходят для этой цели. Однако HTML предназначен для форматирования и представления данных, а не для их определения и верификации. HTML является подмножеством SGML, но не включает конструкций для верификации данных, предусматриваемых спецификацией SGML. Причина этого заключается в том, что SGML очень сложен и замысловат, а его полная реализация может обойтись довольно дорого. По крайней мере, на раннем этапе HTML не касался вопросов верификации данных.

С другой стороны, XML все же *касается* вопросов верификации данных. Этот язык был определен в 1997 году как подмножество SGML. XML намного более строг в плане своего формата, чем HTML, и предназначен для представления данных. XML — не проприетарный язык. Консорциум Всемирной паутины (W3C) — это организация, которая выносит рекомендации и способствует распространению соответствующих стандартов. При описании многих веб-технологий я люблю ссылаться на сайт W3schools (http://www.w3schools.com/).

В последующих главах вы увидите, как XML используется в рамках разных объектно-ориентированных технологий, таких как веб-службы, распределенные вычисления, постоянство объектов и т. д. А в этой главе мы рассмотрим концепции и синтаксис, стоящие за XML (и JSON).

Одна из философских проблем с языком Java состоит в том, что он является проприетарным (принадлежит компании Oracle). Фреймворк .NET тоже является проприетарным (им владеет компания Microsoft), равно как и Objective-C (принадлежит компании Apple). Вся прелесть XML заключается в том, что он представляет собой открытую технологию. Кроме того, этот язык является одним из немногих, которые используются большинством лидеров индустрии информационных технологий. Таким образом, XML не собирается «уходить со сцены» в ближайшее время.

XML в противопоставлении с HTML

Вскоре после появления XML начали ходить слухи о том, что он заменит собой HTML. Многие верили им, поскольку оба языка являются потомками SGML, а XML представляет собой его модернизацию. На самом деле HTML и XML предназначены для разных целей. HTML, как и CSS (каскадные таблицы стилей), определяет структуру и представление документов, в то время как XML описывает данные. Таким образом, оба являются важными инструментами при разработке веб-систем.

XML во многим схож с HTML. Это неудивительно, поскольку они происходят из одного источника. Однако у XML есть основное преимущество, которое отсутствует у HTML, — проверка, валидны ли документы и корректно ли они сформированы.

Все HTML-теги предопределены. Все теги, например `<HTML>`, `<HEAD>`, `<BODY>` и т. д., описаны в спецификации HTML. Вы не можете добавлять свои теги. Поскольку HTML предназначен для форматирования, это не является проблемой. Однако XML призван определять данные. Для этого вам потребуется придумать собственные имена тегов. Именно здесь в дело вступает *определение типа документа (DTD)*. Определение типа документа — это место для задания тегов, которые описывают соответствующие данные. При создании XML-документа можно использовать только предопределенные теги. Все XML-документы проходят проверку на валидность. XML-процессор считывает определение типа документа и решает, является ли этот документ валидным. Если документ не оказывается валидным, то генерируется сообщение о синтаксической ошибке.

ВАЛИДНЫЕ ДОКУМЕНТЫ

Вы не обязаны использовать определения типов документов. Однако их применение обеспечивает значительное преимущество в виде проверки XML-документов на валидность. Без определения типа документа в XML-коде будет проводиться только проверка на предмет того, корректно ли сформирован документ. Вам потребуется явным образом включить определение типа документа, чтобы он был проверен на валидность. Соответствующие параметры указываются в DTD.

Например, если вы создаете систему заказов на покупку, то вам может потребоваться указать тег `<PurchaseOrder>` в определении типа документа. Если вы при этом допустите ошибку в написании имени тега, указав, например, `<PurchasOrder>`, то она будет выявлена и ваш документ окажется помечен как невалидный.

Успешное прохождение документами валидации означает их намного бо́льшую надежность, которая необходима, когда дело касается данных. Например, HTML включает много парных тегов вроде <FONT> и </FONT>. Если вы забудете указать закрывающий тег </FONT>, то браузер все равно загрузит документ, однако результат может оказаться непредсказуемым. В случае с HTML будет сделано наиболее вероятное предположение и работа продолжится. Однако в случае с XML при использовании определения типа документа этого не произойдет. Если окажется, что документ не сконструирован должным образом, то будет сгенерировано сообщение об ошибке и этот документ не будет считаться валидным.

XML и объектно-ориентированные языки программирования

XML работает рука об руку с объектно-ориентированными языками программирования, чтобы обеспечивать возможность переноса информации. Часто бывает так, что то или иное приложение, для написания которого используется объектно-ориентированный язык программирования, разрабатывается для взаимодействия с XML. Чтобы проиллюстрировать это, снова обратимся к примеру, приводившемуся ранее в текущей главе. Компания «Альфа», в частности ее универсальный магазин, использует базу данных Oracle, а компания «Бета», выпускающая пылесосы, работает с базой данных SQL Server. Компания «Альфа» желает приобрести несколько пылесосов у компании «Бета», чтобы затем внести их в свою инвентарную ведомость. Все транзакции будут осуществляться через Интернет.

Коротко говоря, проблема состоит в том, что соответствующие данные хранятся в абсолютно разных базах данных. Даже если бы эти базы данных были одинаковыми, форматы записей в них, скорее всего, отличались бы. Таким образом, необходимо, чтобы компании «Альфа» и «Бета» смогли обмениваться данными, что означает обмен информацией между их базами данных. Однако это не означает прямое физическое соединение между этими базами данных; вопрос здесь состоит в том, как вести деловые операции, — например, одна компания отправляет заказ на покупку, а принимающая компания обрабатывает его.

ПРОПРИЕТАРНЫЕ РЕШЕНИЯ

Мы могли бы создать проприетарное приложение для обеспечения коммуникаций между компаниями «Альфа» и «Бета». Хотя такой подход оправдан, предпочтительнее найти более общее решение (каким и является объектно-ориентированный способ). Например, компания «Альфа» может занимать на рынке положение, позволяющее ей требовать от всех поставщиков соблюдения ее спецификации заявок на покупку. Вот где блистает XML. Компания «Альфа» может создать XML-спецификацию, доступ к которой будет у всех ее поставщиков.

Чтобы добиться цели, которая состоит в соединении систем двух фирм, компания «Альфа» может создать XML-спецификацию, описывающую то, какая информация необходима для завершения транзакций и сохранения сведений в ее базе данных. Именно здесь в дело вступают объектно-ориентированные языки

программирования. Такие языки, как Java, VB или C#, можно использовать для написания кода, извлекающего информацию из базы данных SQL Server компании «Альфа» и создающего XML-документ на основе согласованных стандартов.

Этот XML-документ затем можно будет отправить по Интернету в компанию «Бета», которая воспользуется согласованным XML-стандартом для извлечения информации, содержащейся в XML-документе, и введет ее в базу данных Oracle. На рис. 11.2 демонстрируется поток данных из одной базы данных в другую. Можно увидеть, что данные извлекаются из базы данных SQL приложением/парсером, после чего передаются по сети другому приложению/парсеру. Затем этот парсер преобразует переданную ему информацию в данные в формате Oracle.

Рис. 11.2. Передача данных от приложения к приложению

ПАРСЕРЫ

Парсер — это приложение, которое считывает документ и извлекает определенную информацию. Например, компилятор содержит парсер. Парсер считывает каждую строку кода программы и применяет определенные грамматические правила для выяснения того, как был написан код. Парсер, к примеру, убедился бы в том, что оператор print написан с использованием соответствующего синтаксиса.

Обмен данными между двумя компаниями

На данном этапе целесообразно частично претворить в жизнь наш пример сотрудничества между компаниями «Альфа» и «Бета». Рамки этого исследования предполагают создание XML-документа, который будет содержать сведения о простой транзакции между двумя компаниями. В этом примере мы создадим простой документ, который будет включать информацию, содержащуюся в табл. 11.1. В этой таблице указаны данные, которые будут передаваться из одной компании в другую.

Таблица 11.1. Спецификация для данных, подлежащих передаче

Объект	Категория	Поле
supplier	name	<companyname>
	address	<street>
		<city>
		<state>
		<zip>
	product	<type>
		<price>
		<count>

Валидация документа с определением типа документа (DTD)

В этом примере мы будем отправлять XML-документ из компании «Бета» в компанию «Альфа». Данный XML-документ будет хранить сведения о транзакции, включающие название компании, ее адрес и информацию об определенном товаре. Следует отметить, что эта информация будет вложенной. Другими словами, общим документом, который можно описать как объект, будет документ `supplier`, а вложенное обозначение поставщика будет включать название компании, ее адрес и информацию о товаре. Заметьте, что также будет иметься информация, вложенная в обозначения адреса и товара. Прежде чем двигаться дальше, укажем определение типа документа, которое будет «управлять» всеми транзакциями в этом примере. Соответствующее определение типа документа приведено в листинге 11.1.

Листинг 11.1. Документ определения данных для валидации

```
<!-- Определение типа документа для документа supplier -->
<!ELEMENT supplier ( name, address)>
<!ELEMENT name ( companyname)>
<!ELEMENT companyname ( #PCDATA)>
<!ELEMENT address ( street+, city, state, zip)>
<!ELEMENT street ( #PCDATA)>
<!ELEMENT city ( #PCDATA)>
<!ELEMENT state ( #PCDATA)>
<!ELEMENT zip ( #PCDATA)>
```

Определение типа документа характеризует, как XML-документ сконструирован. В его состав входят теги, которые очень похожи на HTML-теги. Первая строка — это XML-комментарий:

```
<!-- Определение типа документа для документа supplier -->
```

XML-комментарии выполняют ту же функцию, что и комментарии в любом другом языке программирования, — они используются для документирования кода. Как и любой язык, XML предоставляет комментарии для того, чтобы сделать документ более легким для чтения и понимания. Не включайте в него слишком много комментариев, иначе документ окажется трудным для чтения. Рассматриваемый нами документ содержит только один комментарий.

Остальные строки определяют структуру XML-документа. Взглянем на первую строку:

```
<!ELEMENT supplier ( name, address, product)>
```

В этом теге определяется элемент `supplier`. Как было указано в приведенном чуть ранее определении типа документа, `supplier` включает `name`, `address` и `product`. Таким образом, когда XML-парсер будет фактически разбирать XML-документ, этим документом должен оказаться `supplier`, включающий `name`, `address` и `product`.

Переходя на следующий уровень, мы видим, что в состав элемента `name` входит другой элемент с именем `<companyname>`:

```
<!ELEMENT name ( companyname)>
```

Элемент <companyname> затем определяется как элемент данных, обозначенный #PCDATA:

```
<!ELEMENT companyname ( #PCDATA)>
```

Этот тег завершает иерархию дерева элементов. Соответствующее определение типа документа будет называться supplier.dtd. Создать определение типа документа можно в любом текстовом редакторе. На рис. 11.3 приведено окно программы Блокнот, в котором показано, как может выглядеть определение типа документа для нашего приложения.

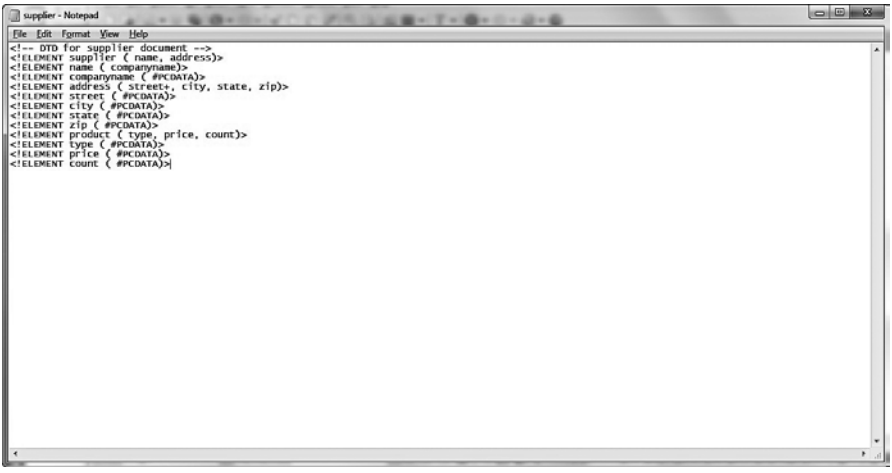

Рис. 11.3. Создание определения типа документа в программе Блокнот

ВАЛИДНОСТЬ ДОКУМЕНТОВ

XML-документ, для которого указано определение типа документа, будет признан либо валидным, либо невалидным, исходя из этого определения типа документа. Если DTD отсутствует, то XML-документ не будет оцениваться как либо валидный, либо невалидный. Надо отметить, что определение типа документа может быть указано внутренне или внешне. Поскольку внешние определения типов документов гораздо эффективнее работают, здесь мы будем использовать именно такое DTD.

```
<!-- Определение типа документа для документа supplier -->
<!ELEMENT supplier ( name, address)>
<!ELEMENT name ( companyname)>
<!ELEMENT companyname ( #PCDATA)>
<!ELEMENT address ( street+, city, state, zip)>
<!ELEMENT street ( #PCDATA)>
<!ELEMENT city ( #PCDATA)>
<!ELEMENT state ( #PCDATA)>
<!ELEMENT zip ( #PCDATA)>
<!ELEMENT product ( type, price, count)>
<!ELEMENT type ( #PCDATA)>
<!ELEMENT price ( #PCDATA)>
<!ELEMENT count ( #PCDATA)>
```

PCDATA

PCDATA означает «разбираемые символьные данные» и является стандартной символьной информацией из определенного текстового файла, подвергающейся разбору. Любые числа, например целочисленные значения, потребуется преобразовать с помощью парсера.

Включение определения типа документа в XML-документ

Теперь, когда мы создали определение типа документа, пора создать фактический XML-документ. Помните, что он должен соответствовать определению типа документа, которое мы только что написали.

В табл. 11.2 указаны некоторые из фактических данных, которые будут содержаться в XML-документе. Опять-таки обратите внимание, что данные содержатся только в конечных элементах, а не в агрегированных вроде `address` и `name`.

Таблица 11.2. Добавление значений в таблицу

Объект	Категория	Поле	Значение
supplier	name	<companyname>	Компания «Бета»
	address	<street>	Онтарио-Стрит, 12000
		<city>	Кливленд
		<state>	Огайо
		<zip>	24388
	product	<type>	Пылесос
		<price>	50.00
		<count>	20

Чтобы ввести эту информацию в XML-документ, мы можем воспользоваться текстовым редактором, как поступили при работе с определением типа документа. Однако, как вы увидите позднее, специально для этой цели были созданы соответствующие инструменты. На рис. 11.4 показан XML-документ, написанный с использованием программы Блокнот. Этот документ носит имя `beta.xml`.

Рис. 11.4. XML-документ компании «Бета» с определением типа документа

Обратите внимание, что вторая строка привязывает этот документ к supplier.dtd, определенному нами ранее:

```
<!DOCTYPE supplier SYSTEM "supplier.dtd">
```

Глядя на рис. 11.4, мы видим, что теговая структура соответствует спецификации. Важно понимать, что теги являются вложенными и лишь конечные теги содержат какие-либо данные. Некоторые из тегов, по сути, являются высокоуровневыми. В какой-то мере все это похоже на концепцию абстрактных классов. Вы можете представлять себе тег <address> как «абстрактный», поскольку мы не определяем его на самом деле. Однако тег <street> можно считать «конкретным» в связи с тем, что мы действительно присваиваем ему значение. Другими словами, тег <street> содержит информацию, в отличие от тега <address>:

```
<address>
<street>Онтарио-Стрит, 12000</street>
```

Есть лучший способ инспектирования XML-документа. Как уже отмечалось ранее, для помощи при разработке XML-документов было создано много соответствующих инструментов. Один из первых инструментов такого рода называется XML Notepad и обладает интерфейсом, как у программы Блокнот, предусмотренной в операционных системах компании Microsoft.

XML NOTEPAD

Вы сейчас можете отыскать эту программу, просто введя в интернет-поисковике словосочетание XML Notepad. Она доступна на разных сайтах.

XML Notepad может помочь понять структуру XML-документа. Установив XML Notepad, вы сможете открыть файл beta.xml. На рис. 11.5 показано, что будет, когда вы откроете файл beta.xml в XML Notepad. Когда документ откроется, разверните все узлы, щелкнув кнопкой мыши на плюсах, чтобы взглянуть на все элементы.

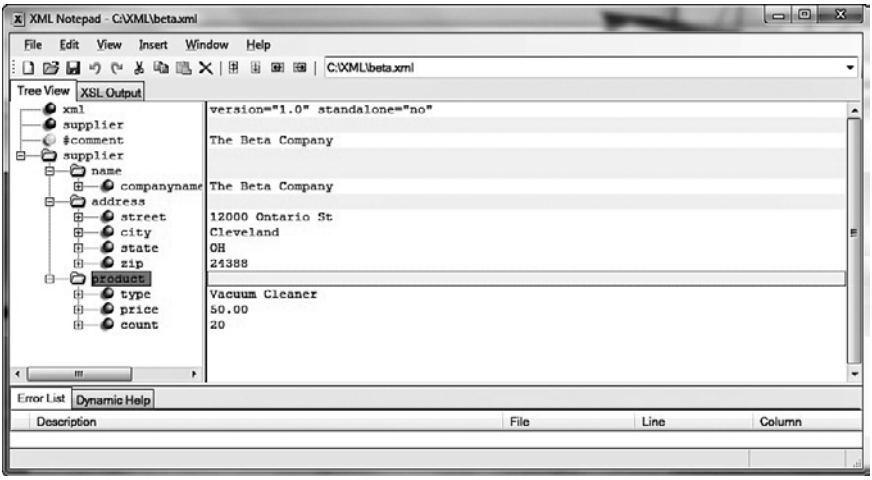

Рис. 11.5. Открытие файла beta.xml в XML Notepad

XML Notepad показывает каждый уровень документа, начиная с тега <supplier>. Обратите внимание, что, как мы уже отмечали ранее, лишь конечные элементы содержат какие-либо данные.

Очевидным преимуществом разработки определения типа документа является то, что оно может быть использовано для нескольких документов — в данном случае для нескольких supplier. Допустим, у нас имеется компания, которая выпускает коньки. Она называется «Гамма» и желает стать поставщиком компании «Альфа». Компании «Гамма» необходимо создать XML-документ, соответствующий supplier.dtd. На рис. 11.6 показан этот документ, открытый в XML Notepad.

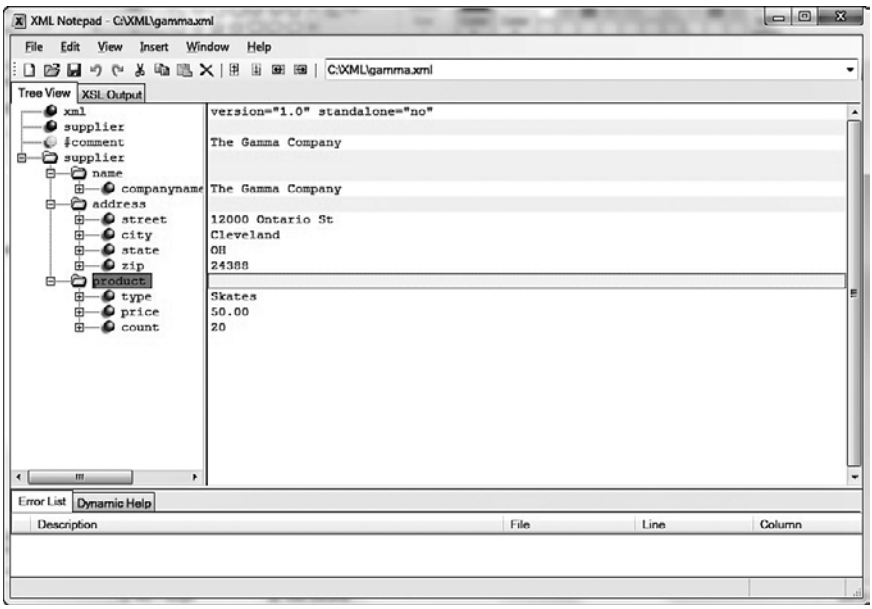

Рис. 11.6. Файл gamma.xml, открытый в XML Notepad

Обратите внимание, что beta.xml и gamma.xml соответствуют supplier.dtd. Вопрос состоит в следующем: что произойдет, если окажется, что XML-документ не соответствует supplier.dtd? На данном этапе проявляется мощь определения типа документа. Умышленно допустим ошибку в файле gamma.xml, удалив всю информацию, касающуюся названия:

```
<name>
<companyname>Компания «Гамма»</companyname>
</name>
```

По сути мы создали невалидный документ — невалидный согласно supplier.dtd. Валидный документ показан на рис. 11.7. Знайте, что программа Блокнот не станет уведомлять вас о том, что документ является невалидным, поскольку она не проводит проверку на валидность. Вам потребуется воспользоваться XML-валидатором, чтобы провести такую проверку.

Рис. 11.7. Невалидный документ (отсутствует информация о названии)

Теперь у нас имеется невалидный документ, исходя из `supplier.dtd`. Как нам убедиться в том, что он является невалидным? Мы можем открыть документ `gamma.xml` с помощью XML-валидатора, например того, что есть на сайте W3schools (www.w3schools.com/xml/xml_validator.asp). Обратите внимание на результат, показанный на рис. 11.8. В данном случае XML-валидатор выдал сообщение, в котором сказано, что был обнаружен невалидный документ.

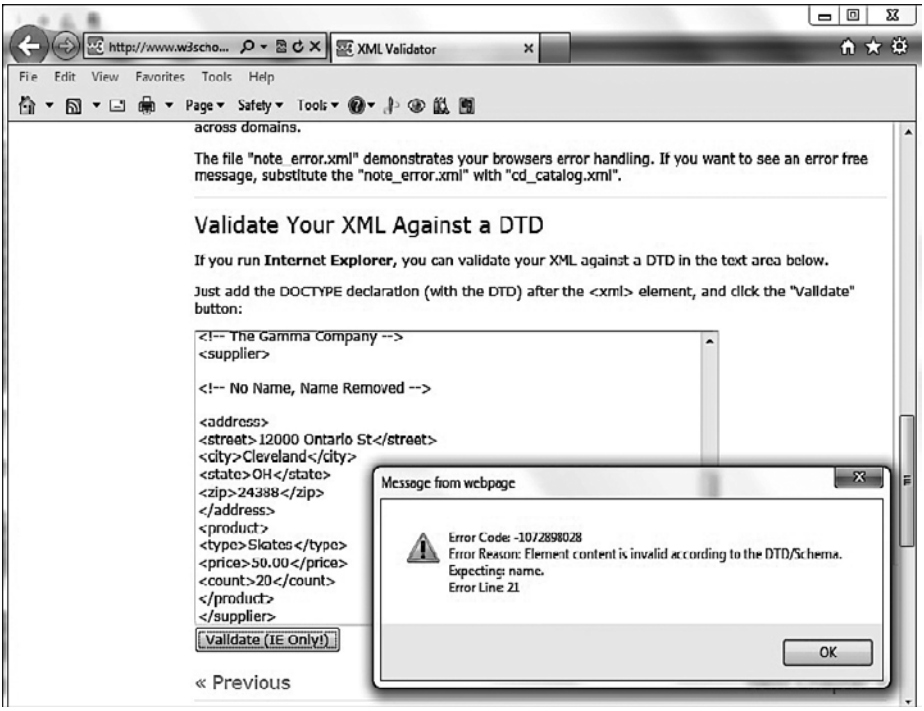

Рис. 11.8. Невалидный документ gamma.xml, проверяемый на сайте W3schools

XML-ВАЛИДАЦИЯ

Валидацию XML-кода позволяют проводить многие приложения. XML-валидатор на сайте W3schools — один из наиболее доступных инструментов такого рода.

Поскольку ожидалось, что документ будет соответствовать определению типа документа `supplier.dtd`, было сгенерировано сообщение об ошибке. Фактически в нем конкретно говорится, в чем заключается проблема. Согласно определению типа документа ожидалось наличие информации, касающейся названия. Таким образом, чтобы создать надлежащий XML-документ для этой системы, потребуется указать всю соответствующую информацию, причем в подходящем формате. Чтобы документ получился валидным, должен быть предусмотрен тег `<address>`.

Здесь необходимо учитывать, что проверка на наличие ошибок не осуществлялась бы в случае с HTML. Кроме того, даже если структура не окажется валидной, вы все равно сможете открыть такой XML-файл в браузере, как показано на рис. 11.9.

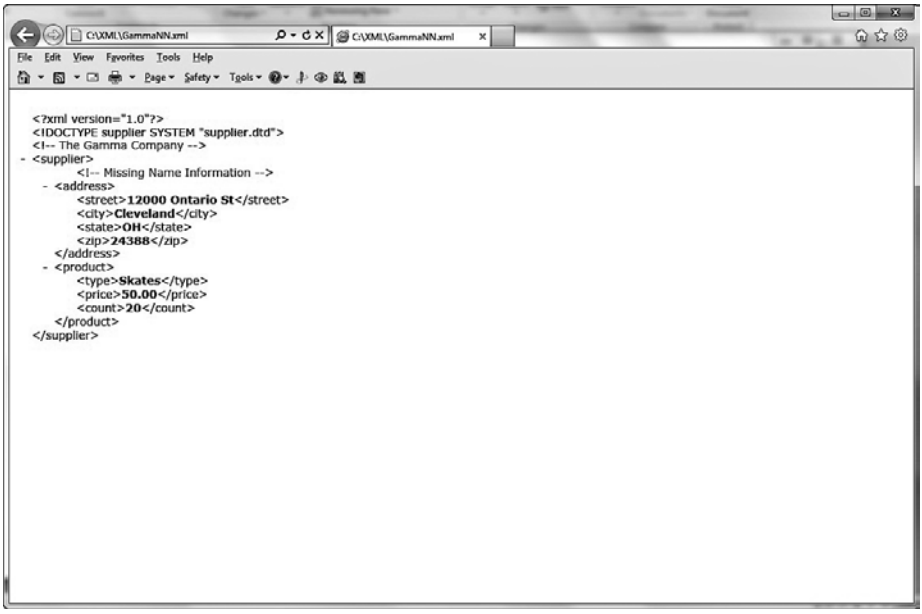

Рис. 11.9. Документ gamma.xml (без информации о названии), открытый в Internet Explorer

Обратите внимание, что, хотя документ является невалидным, браузер открывает и даже отображает его. Так происходит потому, что браузер не проверяет, что документ соответствует требуемому определению типа документа, однако XML-валидаторы все же делают это. Теоретически это одно из основных преимуществ, которые обеспечивает XML во время работы с данными (при использовании валидатора). В то время как HTML используется для *отображения* данных, XML

применяется для *форматирования* данных, а это означает, что при работе с ним приходится проявлять немного большую бдительность. Это важное отличие.

У вас может возникнуть вопрос: какая польза от инструмента вроде XML Notepad в примере с `supplier` и для чего он вообще используется? Вот ответ на первую часть этого вопроса: программа XML Notepad или какой-нибудь похожий на нее текстовый редактор позволяет проверить, является ли определенный документ валидным, на раннем этапе процесса. Ответ на вторую часть этого вопроса звучит так: XML Notepad или подобный редактор можно использовать для конструирования документов.

Использование CSS

С технической точки зрения концепция переносимых данных часто сконцентрирована на перемещении информации между двумя точками. Однако перемещение данных из точки А в точку Б не будет иметь реальной ценности, если данные не окажутся представлены надлежащим образом. Поэтому нам необходимо принимать во внимание то, как данные, перемещаемые в XML-системе, будут представлены пользователям.

Помните, что хотя XML применяется главным образом для определения данных, HTML, по сути является механизмом их представления. Однако XML и HTML можно использовать в тандеме для представления данных в браузерах.

Несмотря на то что XML не предназначен для представления информации, существуют способы форматирования XML-данных. Один из них заключается в использовании CSS — каскадных таблиц стилей. CSS активно применяются в мире HTML для форматирования содержимого. В некоторой степени CSS можно использовать для форматирования XML-данных. Напомню, что XML-документ `supplier` содержит определения для `<companyname>`, `<street>`, `<city>`, `<state>` и `<zip>`. Допустим, нам нужно отформатировать каждое из этих определений, как показано в табл. 11.3, чтобы все соответствовало спецификации, согласно которой должно выполняться форматирование элементов XML-документа.

Таблица 11.3. CSS-спецификация

Тег	Семейство шрифтов (font-family)	Размер шрифта (font-size)	Цвет (color)	Отображение (display)
<companyname>	Arial; sans serif	24	blue	block
<street>	Times New Roman; serif	12	red	block
<city>	Courier New; serif	18	black	block
<state>	Tahoma; serif	16	gray	block
<zip>	Arial Black; sans serif	6	green	block

Мы можем представить это на CSS с использованием следующей таблицы стилей:

```
companyname{font-family:Arial, sans-serif;
    font-size:24;
    color:blue;
    display:block;}
street {font-family:"Times New Roman", serif;
    font-size:12;
    color:red;
    display:block;}
city {font-family:"Courier New", serif;
    font-size:18;
    color:black;
    display:block;}
state {font-family:"Tahoma"; serif;
    font-size:16;
    color:gray;
    display:block;}
zip {font-family:"Arial Black", sans-serif;
    font-size:6;
    color:green;
    display:block;}
```

Эту таблицу стилей можно подключить, добавив такую строку кода в наш XML-документ:

```
<?xml-stylesheet href="supplier.css" type="text/css" ?>
```

Например, почтовый индекс ранее отображался шрифтом по умолчанию, а теперь отформатирован так: для него установлен шрифт Arial Black с кеглем 6, зеленого цвета. Благодаря указанию свойства `display:block` в данной ситуации каждый атрибут будет располагаться на новой строке.

Вставить этот код можно следующим образом:

```
<?xml version="1.0" standalone="no"?>
<?xml-stylesheet href="supplier.css" type="text/css" ?>
<!DOCTYPE supplier SYSTEM "supplier.dtd">
<!-- XML-данные -->
<supplier>
<name>
<companyname>Компания «Бета»</companyname>
</name>
<address>
<street>Онтарио-Стрит, 12000</street>
<city>Кливленд</city>
<state>Огайо</state>
<zip>24388</zip>
</address>
</supplier>
```

Добавим CSS-код в XML-документ и откроем этот документ в браузере. Результат показан на рис. 11.10.

Снова взгляните на рис. 11.9, чтобы увидеть, как этот документ выглядел без CSS.

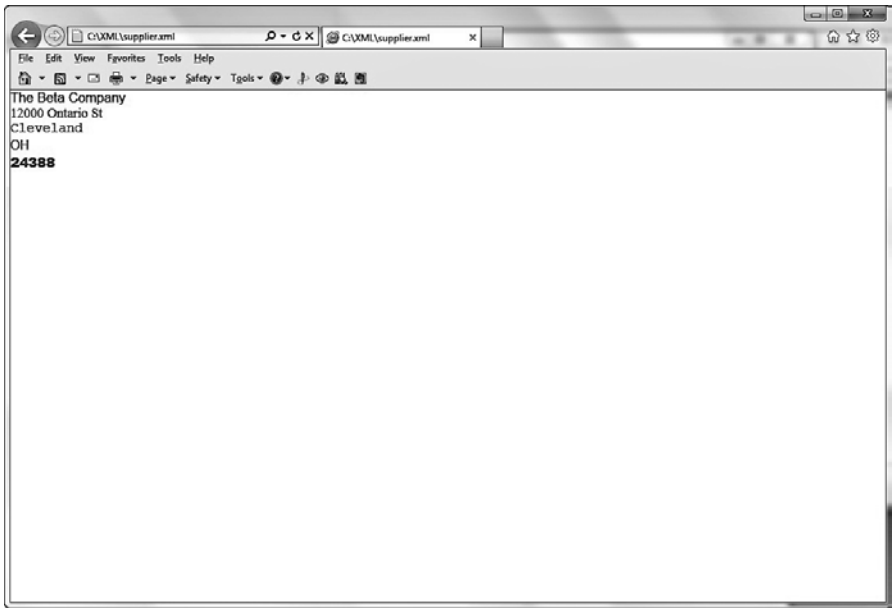

Рис. 11.10. XML-документ с использованием CSS

JavaScript Object Notation (JSON)

Хотя концепция переносимых данных очень эффективна, а XML — это стандарт, широко распространенный в сфере обмена данными, также существуют альтернативные решения. Стремление создать отраслевой стандарт — благородная цель, однако это чрезвычайно трудно сделать в индустрии, в которой постоянно появляются новшества и изменения. Например, многие разработчики считают, что преимущества строго типизированных, компилируемых языков слишком сложно игнорировать, в то время как другие высоко ценят гибкость языков вроде Perl, которые менее типизированы. Дело обстоит аналогичным образом, если вести речь о данных.

Несмотря на то что XML намного более структурирован, особенно при использовании определений типов документов, технологии вроде JavaScript Object Notation (Нотация объектов JavaScript) попадают в категорию «более гибких». На сайте W2schools (http://www.w3schools.com/json/default.asp) приведено следующее описание JSON в виде списка:

- JSON — это облегченный формат обмена текстовыми данными;
- JSON не зависит от языка;
- JSON является «самоописываемым» и легким для понимания.

** JSON задействует JavaScript-синтаксис для описания объектов данных, и тем не менее является независимым от языка и платформы. Существуют JSON-парсеры и JSON-библиотеки для многих разных языков программирования.*

Здесь важно осознавать, что JSON задействует тот же самый синтаксис для создания объектов, что и JavaScript. Таким образом, как разъясняется на сайте

W3shoools, *«вместо использования парсера JavaScript-программа может прибегнуть к встроенной функции eval() и обработать JSON-данные для создания нативных JavaScript-объектов»*. Фактически вы можете преобразовывать эти данные разными путями; некоторые разработчики стараются не использовать функцию `eval()`.

Как и во всех случаях в этой книге, для того чтобы отшлифовать свои знания соответствующих концепций, преобразуем XML-объект, созданный нами ранее в текущей главе, в эквивалентный JSON-образец. Кроме того, взглянем на простой JSON-объект, созданный на сайте W3schools:

```
{
"employees": [
{ "firstName":"Джон" , "lastName":"Доу" },
{ "firstName":"Анна" , "lastName":"Смит" },
{ "firstName":"Питер" , "lastName":"Джонс" }
]
}
```

Обратите внимание, что фактический JSON-объект представлен кодом, который заключен во внешние фигурные скобки. Заметьте также, что JSON-объект по сути является свойством — парой «имя/значение». Чтобы понять, как JSON-объект вписывается в общую картину, взгляните на пример полного кода, который также имеется на сайте W3schools. Вы можете поэкспериментировать, используя редактор, предусмотренный на этом сайте (http://www.w3schools.com/json/json_intro.asp):

```
<!DOCTYPE html>
<html>
 <body>
 <h2>Создание JSON-объекта на JavaScript</h2>
<p>
 Имя: <span id="jname"></span><br />
Возраст: <span id="jage"></span><br />
Адрес: <span id="jstreet"></span><br />
Телефон: <span id="jphone"></span><br />
</p>

<script type="text/javascript">
 var JSONObject= {
 "name":"Джон Джонсон",
 "street":"Осло-Вест, 555",
"age":33,
 "phone":"5551234567"};
document.getElementById("jname").innerHTML=JSONObject.name
document.getElementById("jage").innerHTML=JSONObject.age
document.getElementById("jstreet").innerHTML=JSONObject.street
document.getElementById("jphone").innerHTML=JSONObject.phone
</script>
</body>
</html>
```

JSON-РЕДАКТОР НА САЙТЕ W3SCHOOLS

Следует отметить, что редактор, предлагаемый на сайте W3schools, обрабатывает по одному целому файлу. Таким образом, если вы захотите поработать с отдельным файлом (вроде таблицы стилей и т. д.), вам придется вставить его в редактор. Коротко говоря, вы сможете обрабатывать в этом редакторе только по одному файлу, так как он предназначен для проверки каких-либо концепций. При создании собственного приложения вам потребуется структурировать код так, чтобы он располагался в логически раздельных файлах.

Используя этот пример, реализуем XML-объект, созданный ранее, как JSON-объект:

```
var address= {
 "street":"Мэйн-Стрит, 23456",
 "city":"Кливленд",
 "state":"Огайо",
 "zip":"24388"
 };
```

В данном случае JSON-объект создается как ассоциативный JavaScript-массив с использованием пар «имя/значение», которые упоминались ранее. На рис. 11.11 показано, как выполняется вложение JSON-объекта в фактический HTML-документ.

Рис. 11.11. HTML-документ с использованием JSON-объекта

Чтобы понять, как можно обработать и отобразить этот JSON-код подобно тому, как это было в нашем XML-примере ранее, мы можем воспользоваться следующей таблицей стилей:

```
<!DOCTYPE html>
<html>
```

```
 <head>
 <style type="text/css">
companyname{font-family:Arial, sans-serif;
    font-size:24;
    color:blue;
    display:block;}
street {font-family:"Times New Roman", serif;
    font-size:12;
    color:red;
    display:block;}
city {font-family:"Courier New", serif;
    font-size:18;
    color:black;
    display:block;}
state {font-family:"Tahoma"; serif;
    font-size:16;
    color:gray;
    display:block;}
zip {font-family:"Arial Black", sans-serif;
    font-size:6;
    color:green;
    display:block;}
 </style>
 </head>

 <body>
 <companyname>Компания «Бета»</companyname>

<p>
<street> <span id="jstreet"></span><br /> </street>
<city> <span id="jcity"></span><br /> </city>
<state> <span id="jstate"></span><br /> </state>
<zip> <span id="jzip"></span><br /> </zip>
</p>

<script type="text/javascript">
 var address= {
 "street":"Мэйн-Стрит, 23456",
 "city":"Кливленд",
 "state":"Огайо",
 "zip":"24388"
 };

document.getElementById("jstreet").innerHTML=address.street
document.getElementById("jcity").innerHTML=address.city
document.getElementById("jstate").innerHTML=address.state
document.getElementById("jzip").innerHTML=address.zip
</script>
</body>
</html>
```

Выполнение этого кода в редакторе на сайте W3schools сгенерирует вывод, показанный на рис. 11.12.

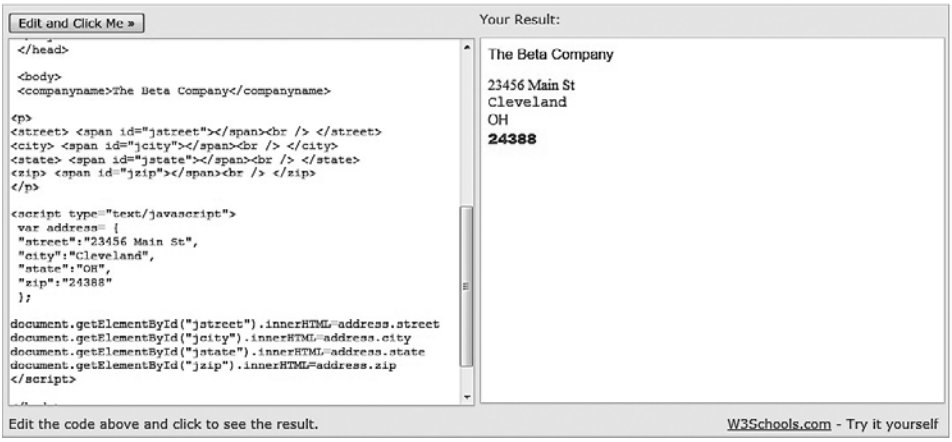

Рис. 11.12. HTML/CSS-документ с использованием JSON-объекта в редакторе на сайте W3schools

И наконец, сравним вывод, который был сгенерирован при выполнении XML-кода в браузере Internet Explorer и приведен на рис. 11.10, с JSON-примером. Для этого мы прибегнем к тому же самому механизму; откройте JSON-файл (вложенный в HTML-код) в Internet Explorer. Соответствующий результат приведен на рис. 11.13.

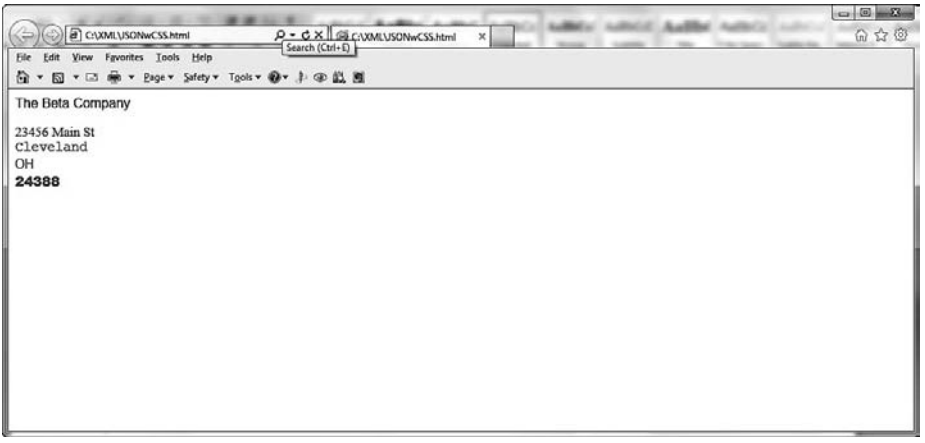

Рис. 11.13. JSON-реализация аналогичного XML-примера

Как вы можете видеть, несмотря на то что XML и JSON по-разному подходят к концепции переносимых данных, цель одна. В обоих случаях она заключается в том, чтобы сделать возможной передачу данных, содержащихся в объектах, которые можно будет легко разобрать, обмениваться ими и использовать их. Как уже

отмечалось ранее, многим людям нравится применять JSON, поскольку он менее структурирован, чем XML, и они при этом утверждают, что он позволяет быстрее решать соответствующие задачи.

Резюме

В этой главе мы рассмотрели многие особенности XML, а также узнали, почему он очень важен для сообщества, занимающегося информационными технологиями. Редко бывает так, что основные игроки на рынке информационных технологий «ведутся» на один и тот же стандарт, однако именно это произошло с XML. Мы также кратко рассмотрели альтернативную технологию под названием JSON. Как XML, так и JSON используются для передачи того, что мне нравится называть *переносимыми данными*.

Исходя из объектно-ориентированной точки зрения, после прочтения этой главы вы должны понять, что объектно-ориентированная разработка простирается далеко за пределы ОО-языков, а также охватывает данные. Поскольку данные представляют собой фундаментальную часть информационных систем, важно проектировать объектно-ориентированные системы, сосредоточенные на данных. В современной бизнес-среде перемещение данных из одной точки в другую имеет первостепенное значение.

Есть много уровней исследования, на которые можно заглянуть, когда дело касается XML и JSON. Эта книга посвящена концепциям, а к концу текущей главы у вас должно было сложиться общее представление о том, что такое XML и JSON, а также о некоторых инструментах, используемых в сочетании с ними. Кроме того, в этой главе кратко упомянут еще один уровень — уровень таблиц стилей. Используя CSS и другие технологии, вы сможете еще лучше отформатировать свои XML-документы.

Ссылки

- *Маркотт Итан*. Отзывчивый веб-дизайн (Responsive Web Design). — Нью-Йорк: A Book Apart, 2011.
- http://www.w3schools.com/json/default.asp.
- *Дейтел Харви и Дейтел Пол*. JavaScript для программистов (JavaScript for Programmers). — Аппер-Сэддл-Ривер: Pearson Education, Inc, 2010.
- *Хьюз Шерил*. Руководство веб-мастера по XML (The Web Wizard's Guide to XML). — Бостон: Addison-Wesley, 2003.
- *Уотт Эндрю Х*. Освой самостоятельно XML за 10 минут (Teach Yourself XML in 10 Minutes). — Индианаполис: Sams Publishing, 2003.
- *Маккиннон Аль и Маккиннон Линда*. XML: серия книг «Web Warrior» (XML: Web Warrior Series). — Бостон: Course Technology, 2003.
- *Хольцнер Стивен*. Реальный XML (Real World XML). — Индианаполис: New Riders, 2003.

Глава 12

Постоянные объекты: сериализация, маршалинг и реляционные базы данных

Независимо от того, бизнес-приложение какого типа вы станете создавать, база данных, скорее всего, будет частью «уравнения». Кстати, одна из моих любимых реплик по поводу разработки программного обеспечения звучит так: «Все дело в данных». Коротко говоря, независимо от того, какие аппаратное обеспечение, операционная система, прикладное программное обеспечение и т. д. используются при разработке приложения, причиной, по которой определенная система вообще создается, обычно является необходимость в данных.

Основные положения, касающиеся постоянных объектов

Напомню, что, когда приложение создает экземпляр объекта, он «проживет» ровно столько, сколько будет выполняться само приложение. Таким образом, если вы создадите экземпляр объекта `Employee`, который содержит атрибуты, например `name`, `ss#` и т. д., то этот объект прекратит свое существование, когда выполнение приложения завершится. На рис. 12.1 проиллюстрирован традиционный жизненный цикл объекта, который довольно прост. Когда приложение создает объект, он «живет» в рамках этого приложения. Когда выполнение приложения завершается, объект оказывается вне области видимости. Чтобы объект продолжил «жить», он должен быть записан в какое-то постоянное хранилище.

Созданный и инициализированный экземпляр объекта `Employee` пребывает в определенном состоянии. Помните, что состояние объекта определяется значениями его атрибутов. Если мы хотим сохранить состояние объекта `Employee`, то нам потребуется предпринять для этого определенные действия. Концепция сохранения состояния объекта с целью его использования позднее называется концепцией *постоянства*. Таким образом, мы употребили термин «постоянный объект» для определения объекта, который может быть восстановлен и использован независимо от того или иного приложения. На рис. 12.2 проиллюстрирован традиционный жизненный цикл объекта с постоянством. На этом рисунке объект создается в приложении 1, которое затем записывает его на «запоминающее устройство», скажем в базу данных. Поскольку в итоге этот объект располагается в постоянном хранилище, другие приложения

могут получить к нему доступ. На этом рисунке также показано, что приложение 2 теперь может создать экземпляр объекта и загрузить содержимое постоянного объекта.

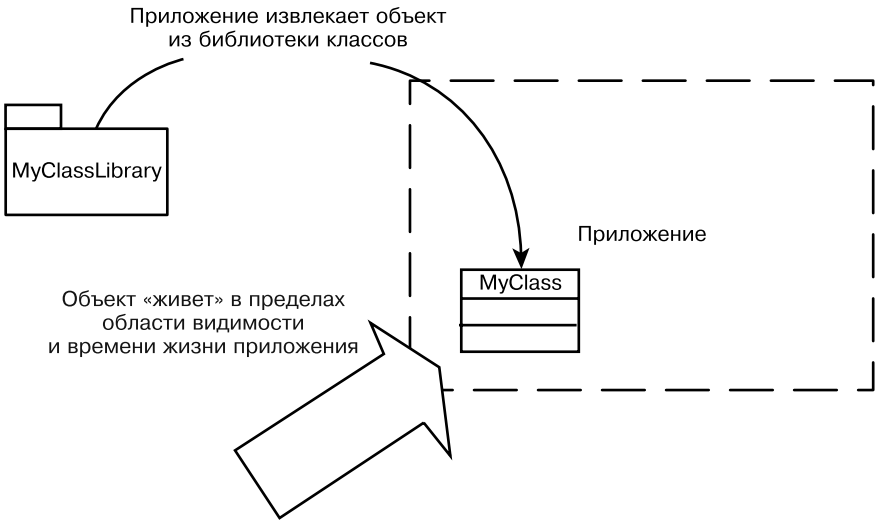

Рис. 12.1. Жизненный цикл объекта

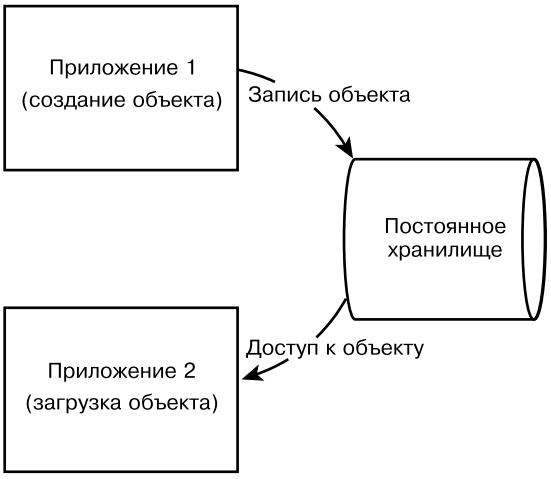

Рис. 12.2. Жизненный цикл объекта с постоянством

Есть много способов сохранения состояния того или иного объекта. Вот некоторые из них:

- сохранение в плоском файле;
- сохранение в реляционной базе данных;
- сохранение в объектной базе данных.

Самый легкий способ продемонстрировать сохранение объекта — создать код, который запишет этот объект в плоский файл, поскольку у многих людей нет доступа к объектным базам данных или промышленным реляционным базам данных с их домашних компьютеров. Однако, несмотря на то что использование плоских файлов хорошо подходит в качестве такого наглядного примера, применение этих файлов в бизнес-приложениях, конечно же, не является нормой.

Сохранение объекта в плоском файле

В этом разделе мы воспользуемся плоским файлом для иллюстрирования постоянства объектов. Я определяю плоский файл как простой файл, управляемый операционной системой. Это очень простая концепция, так что не нужно зацикливаться на ее описании.

ПЛОСКИЕ ФАЙЛЫ

Многие люди не совсем согласны с тем, что термин «плоский файл» удачный. Слово «плоский» подразумевает, что объект буквально делается плоским, что в некотором отношении действительно так и есть. Вы почти можете считать процесс «уплощения» тем, что необходимо для сохранения и перемещения какого-либо объекта, независимо от его сложности.

Одна из проблем, о которых вы, возможно, задумывались, заключается в том, что объект нельзя сохранить в файле как простую переменную — и это правда. Фактически эта проблема сохранения состояний объектов породила крупный сегмент индустрии создания программных продуктов, о котором мы подробно поговорим позднее в этой главе. Обычно при сохранении ряда переменных в файле вам известен их порядок и тип каждой переменной (возможно, с использованием запятых в качестве разделителей и т. п.), которые вы в итоге записываете в файл. Это может быть файл, где в качестве разделителей используются запятые, или любой другой протокол, который вы, возможно, решите реализовать.

Проблема с объектом основана на том, что он является не просто набором примитивных переменных. Объект можно представлять себе как единый блок, состоящий из нескольких частей. Таким образом, должна быть выполнена декомпозиция объекта, чтобы получить блоки, которые можно будет записать в такую среду хранения, как плоский файл. После декомпозиции объекта и его записи в плоский файл останется одна большая проблема, требующая решения, — восстановление объекта, то есть, по сути, его обратная сборка.

Еще одна большая проблема с сохранением объектов связана с тем фактом, что объект может содержать другие объекты. Допустим, объект `Car` включает в себя такие объекты, как `Engines` и `Wheels`. При сохранении этого объекта в плоском файле вы должны понимать, что необходимо сохранить весь объект `Car`, содержащий `Engines` и т. д., целиком.

В современных языках программирования имеются встроенные механизмы обеспечения постоянства объектов. Например, в Java, как и в других основанных на C языках, часто задействуется концепция потока, когда речь идет о вводе/выводе. Сохранить объект в файле при работе с Java можно, записав его в файл

с применением Stream. Для записи в Stream объекты должны реализовывать либо интерфейс Serializable, либо интерфейс Externalizable.

Недостаток этого подхода заключается в том, что применяемое решение является проприетарным — вам придется использовать Java, чтобы все получилось. Фактически язык Java должен быть на обоих концах «канала». Другой, более соответствующий концепции переносимости подход к решению требуемой задачи состоит в создании XML-документа как промежуточного файла и декомпозиции объекта с последующим его восстановлением с применением открытых XML-технологий.

Мы рассмотрим оба подхода в текущей главе. Сначала будет использован язык Java для демонстрации технологии сериализации Java, а затем мы прибегнем к XML-стратегии для реализации .NET-примера с применением C#.

Сериализация файла

В качестве примера взгляните на приведенный далее Java-код для класса Person:

```
package Serialization;
import java.util.*;
import java.io.*;

  class Person implements Serializable{

    private String name;

    public Person(){
  }

  public Person(String n){
    System.out.println("Внутри конструктора для Person");
    name = n;
  }

  String getName() {
   return name;
  }

}
```

Это простой класс, содержащий только один атрибут, который представляет имя соответствующей персоны.

В строке, которой следует уделить особое внимание, определяется класс Serializable. Если вы заглянете в документацию к Java, то поймете, что интерфейс Serializable не содержит большого количества информации — фактически он предназначен исключительно для указания на то, что объект будет сериализован:

```
class Person implements Serializable
```

Этот класс также содержит метод getName, который возвращает имя объекта. В действительности, за исключением интерфейса Serializable, в этом примере нет ничего нового, чего бы мы не видели ранее. Именно здесь начинается интересное.

Теперь нам нужно создать приложение, которое запишет этот объект в плоский файл. Это приложение будет называться SavePerson и выглядеть следующим образом:

```
package Serialization;
import java.util.*;
import java.io.*;
public class SavePerson implements Serializable{

  public SavePerson(){

    Person person = new Person("Джек Джонс");

    try{
      FileOutputStream fos = new FileOutputStream("Name.txt");
      ObjectOutputStream oos = new ObjectOutputStream(fos);
      System.out.print("Записанное имя персоны: ");
      System.out.println(person.getName());

      oos.writeObject(person);
      oos.flush();
      oos.close();
    } catch(Exception e){
      e.printStackTrace();
    }

  }
}
```

Несмотря на то что часть этого кода углубляется в более сложную Java-функциональность, мы можем в общих чертах понять, что происходит, когда объект подвергается сериализации (преобразованию) и записи в файл.

JAVA-КОД

Хотя мы не рассматривали явным образом часть кода в этом образце, например код, касающийся файлового ввода-вывода, вы можете тщательнее изучить его, воспользовавшись книгами, упомянутыми в конце этой главы.

К настоящему времени вы должны понимать, что это уже фактическое приложение. Откуда это известно? Об этом свидетельствует тот факт, что код включает основной метод. По сути это приложение делает три вещи.

1. Создает экземпляр объекта Person.
2. Сериализует этот объект.
3. Записывает этот объект в файл Name.txt.

Акт сериализации и записи объекта совершается в следующем коде:

```
oos.writeObject(person);
```

Это намного проще, чем записывать каждый атрибут поодиночке. Очень удобно записывать объект прямо в файл.

Еще раз о реализации и интерфейсе

Интересно отметить, что основополагающая реализация преобразования файла не так проста, как используемый интерфейс. Помните, что одна из наиболее важных тем этой книги — концепция разделения реализации и интерфейса. Обеспечение интуитивно понятного и простого в применении интерфейса, скрывающего основополагающую реализацию, намного облегчает жизнь пользователям.

Сериализация файла — еще один отличный пример различия между интерфейсом и реализацией. Интерфейс программиста используется для записи объекта в файл. Вас не волнуют все технические тонкости совершения этого «подвига». Для вас важно лишь следующее:

- вы сможете записать файл как единый блок;
- вы сможете восстановить объект точно так же, как вы его сохранили.

Это то же самое, что пользоваться автомобилем. Для того чтобы завести автомобиль, вы применяете интерфейс — ключ в замке зажигания, который позволяет запустить двигатель. Большинству людей неизвестны или безразличны технические моменты того, как все это работает, — для них важно лишь то, что автомобиль заводится.

Программа `SavePerson` записывает объект в файл `Name.txt`. Приведенный далее код восстанавливает этот объект:

```
package Serialization;
import java.io.*;
import java.util.*;

public class RestorePerson{

  public RestorePerson(){
    try{
      FileInputStream fis = new FileInputStream("Name.txt");
      ObjectInputStream ois = new ObjectInputStream(fis);

      Person person = (Person )ois.readObject();
      System.out.print("Восстановленное имя персоны: ");
      System.out.println(person.getName());
      ois.close();
    } catch(Exception e){
      e.printStackTrace();
    }

  }
}
```

Больше всего представляет интерес код, который извлекает объект из файла `Name.txt`:

```
Person person = (Person )ois.readObject();
```

Важно отметить, что объект реконструируется из плоского файла, при этом создается и инициализируется новый экземпляр `Person`. Этот объект `Person`

является точной копией Person, сохраненного нами с использованием приложения SavePerson. На рис. 12.3 показан вывод как приложения SavePerson, так и RestorePerson.

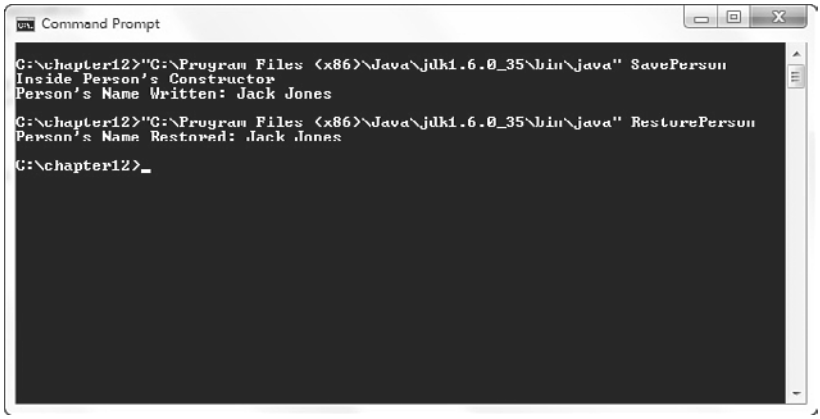

Рис. 12.3. Сериализация объекта

Обратите внимание, что приведенное на рис. 12.3 имя Jack Jones, которое является частью объекта Person, сохраняется в файле Name.txt при выполнении соответствующего кода, а затем объект восстанавливается при выполнении RestorePerson. Когда объект будет восстановлен, мы сможем получить доступ к атрибуту Person.

А как насчет методов?

Вопрос, который может возникнуть у вас при разговоре о постоянстве объектов, звучит так: «При сохранении объекта легко представить себе, каким образом сохраняются атрибуты, а как насчет методов?»

Одно из определений объекта состоит в том, что он содержит атрибуты и поведения, или, другими словами, данные и методы. Что происходит с методами при сохранении объекта?

В примере сериализации Java методы не сохранялись явным образом. Помните, что, как отмечалось ранее, Java должен быть на обоих концах «канала». В действительности используемые вами определения классов тоже должны быть на обоих концах «канала».

Таким образом, в примере с объектом Person у обоих приложений SavePerson и RestorePerson должен иметься доступ к классу Person. Несмотря на то что возможен динамический доступ к классу Person, у приложения, которое будет использовать этот класс, должен иметься стандартный доступ к нему. Поэтому методы как таковые не обязательно держать в хранилище данных.

Однако, с точки зрения программиста, атрибуты и поведения все равно инкапсулируются как часть объекта. Какие-либо концептуальные отличия отсутствуют, несмотря на то что физическая реализация может не соответствовать концептуальной модели.

Использование XML в процессе сериализации

Несмотря на то что использование того или иного проприетарного решения для выполнения сериализации может оказаться эффективным и быстрым подходом, переносимость будет отсутствовать. XML — это стандарт для определения данных, поэтому мы можем создать XML-модель нашего примера сериализации, которую, по крайней мере теоретически, можно будет использовать при работе с разными платформами и языками. Доступ к XML-модели, которую мы создадим в этом разделе, будет возможен благодаря коду, написанному на C# .NET. Кроме того, ничто не помешает нам получить доступ к сгенерированному XML-файлу из программы, написанной на Java или любом другом языке.

Основное отличие XML-модели от модели сериализации Java заключается в том, что в первом случае мы генерируем XML-документ. Этот документ представляет атрибуты и свойства класса `Person`. Такой подход немного усложняет класс `Person`; вместе с тем соответствующий синтаксис позволяет сделать конструкцию этого класса более инкапсулированной.

Сначала взглянем на C#-код. Основное отличие класса `Person` состоит в способе определения атрибутов. Хотя значительная часть кода аналогична коду в случае с моделью, не основанной на XML (например, конструкторы, поведения и т. д.), данные определяются с учетом XML.

Скажем, вы добавляете определения `XmlRoot`, `XmlAttribute` и `XmlElement` прямо в код. Они будут выглядеть следующим образом:

```
  [XmlRoot("person")]
  public class Person
...
  [XmlAttribute("name")]
  public String Name
...
  [XmlElement("age")]
  public int Age
```

Интересным дополнением к этой стратегии является то, что у самих атрибутов имеются специфические свойства. Хотя такая особенность может потребовать дополнительных строк кода и, следовательно, повысить сложность, она несет в себе серьезное преимущество — инкапсуляция класса получается более сильной. Например, по ходу всей этой книги часто отмечаются преимущества закрытых атрибутов, а также то, как должен осуществляться доступ к этим атрибутам с помощью определенных *геттеров* и *сеттеров*. Ясно, что это веская и важная концепция, но факт остается фактом — определение (и, следовательно, подписи) геттеров и сеттеров предоставляется на усмотрение программиста. Коротко говоря, геттеры и сеттеры могут быть определены с использованием любых имен методов, которые придут в голову программисту. В этой XML-модели геттеры и сеттеры являются свойствами атрибута, поэтому привязываются к нему стандартным образом.

Например, при создании XML-атрибута с именем `Name` определение будет выглядеть так:

```
[XmlAttribute("name")]
  public String Name
  {
    get
    {
      return this.strName;
    }
    set
    {
      if (value == null) return;
      this.strName = value;
    }
  }
```

Взглянув на эти строки, мы видим, что здесь имеется намного больше кода, чем при простом объявлении атрибута:

```
public String Name;
```

Однако, несмотря на то что мы определили атрибут как имеющий тип `String`, теперь атрибут `Name` определен как XML-атрибут, а соответствующие геттер и сеттер являются свойствами атрибута `Name` как такового.

Валидация и верификация данных по-прежнему осуществляются точно так же; вместе с тем все это стало намного понятнее (по крайней мере после того, как вы осознаете это).

Синтаксис для присвоения значения атрибуту `Name` теперь превращается в простой оператор присваивания, как в этой строке кода:

```
this.Name = name;
```

При выполнении этой строки кода вызывается свойство `set` атрибута. По сути здесь происходит перегрузка оператора (для тех, кому доводилось много программировать на C и C++). Когда оператор присваивания (знак равенства) стоит возле атрибута `Name` (слева от него), происходит вызов геттера. Это почти как директива компилятора `inline`.

Концепция использования XML-версии класса `Person` схожа с моделью сериализации Java. Вот образец кода:

```
public void Serialize()
{
  Person[] myPeople = new Person[3];
  myPeople[0] = new Person("Джон Кью Паблик", 32, 95);
  myPeople[1] = new Person("Джейкоб М. Смит", 35, 67);
  myPeople[2] = new Person("Джо Л. Джонс", 65, 77);
  XmlSerializer mySerializer = new XmlSerializer(typeof(Person[]));
  TextWriter myWriter = new StreamWriter("person.xml");
  mySerializer.Serialize(myWriter, myPeople);
  myWriter.Close();
}
```

Основное отличие заключается в том, что вместо сериализации в проприетарный Java-формат файл создается в формате XML:

```
<?xml version="1.0" encoding="utf-8"?>
<ArrayOfPerson xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema">
  <Person name="Джон Кью Паблик">
    <age>32</age>
  </Person>
  <Person name="Джейком М. Смит">
    <age>35</age>
  </Person>
  <Person name="Джо Л. Джонс">
    <age>65</age>
  </Person>
</ArrayOfPerson>
```

Для восстановления объекта мы воспользуемся приведенным далее кодом:

```
public void DeSerialize()
{
  Person[] myRestoredPeople;
  XmlSerializer mySerializer = newXmlSerializer(typeof(Person[]));
  TextReader myReader = new StreamReader("person.xml");
  myRestoredPeople = (Person[])mySerializer.Deserialize(myReader);
  Console.WriteLine("Мои восстановленные люди:");
  foreach (Person listPerson in myRestoredPeople)
  {
    Console.WriteLine(listPerson.Name + " имеет возраст " + listPerson.Age
                                                          + " лет.");
  }
  Console.WriteLine("Нажмите любую клавишу, чтобы продолжить...")
  Console.ReadKey();
}
```

Обратите внимание, что мы осуществляем итерацию по структуре данных с использованием цикла `foreach`. Полный код этого примера на C# приведен в конце текущей главы.

Как вы уже заметили, одно из основных преимуществ этого подхода состоит в том, что XML-файл оказывается доступным при использовании любых языков и платформ, которые реализуют XML-интерфейс, включая Java. Хотя мы и реализовали Java-образец с использованием проприетарного решения, это было сделано для примера. Ничто не помешает программистам также прибегнуть к XML-подходу при работе с Java.

Запись в реляционную базу данных

Реляционные базы данных, пожалуй, представляют собой один из наиболее успешных инструментов, когда-либо изобретенных в области информационных технологий. Хотя некоторые люди, возможно, не согласятся с этим высказыванием, реляционные базы данных оказали огромное влияние на индустрию информационных технологий. Фактически они продолжают оставаться движущей силой, несмотря на то что другие решения вполне могут быть технологически лучше их.

Причина заключается в том, что большинство компаний предпочитают использовать именно реляционные базы данных. Они применяются повсюду — от Oracle до SQL Server в случае с крупными приложениями и Microsoft Access, если речь идет о приложениях небольшой и средней величины.

Несмотря на то что реляционные базы данных так замечательны, они создают некоторые проблемы, когда дело касается взаимодействия с объектами. Как и в случае с задачей записи в плоский файл, может оказаться проблематичным взять объект, который может состоять из других объектов, и записать его в реляционные базы данных, не спроектированные объектно-ориентированным путем.

Реляционные базы данных основаны на концепции таблиц. На рис. 12.4 показаны типичные отношения между таблицами Microsoft Access. Эта реляционная модель настолько широко распространена, что многие люди интуитивно представляют себе все модели данных таким образом. Однако объектно-ориентированная модель не управляется таблицами. На рис. 12.4 показана привычная реляционная модель Northwind, созданная в Microsoft Access.

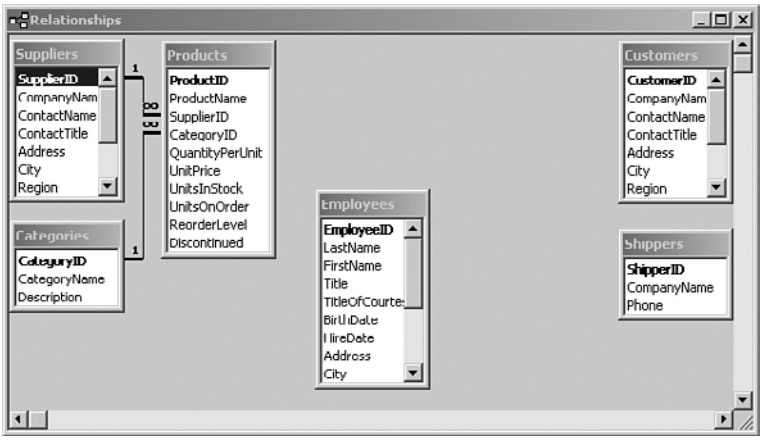

Рис. 12.4. Реляционная модель

Поскольку объекты не отображались удобно в таблицах, в 1990-х годах были разработаны объектно-ориентированные системы баз данных. Хотя эти базы данных хорошо отражали объектно-ориентированную модель и даже могли похвастаться лучшей производительностью, была одна большая проблема: унаследованные данные.

УНАСЛЕДОВАННЫЕ ДАННЫЕ

Унаследованные данные могут представлять собой десятки наборов данных, хранящихся на разных запоминающих устройствах. В этой главе мы рассматриваем унаследованные данные как исторические данные, располагающиеся в реляционных базах данных. Многим людям не нравится слово «унаследованные», поскольку они считают, что оно подразумевает «устаревшие». Фактически важные унаследованные данные не являются устаревшими и представляют собой существенную часть системы.

Поскольку многие компании используют реляционные базы данных, большинство современной бизнес-информации хранится именно в таких базах данных. Это означает, что в эти базы данных были вложены огромные инвестиции. Немаловажна еще одна особенность этих систем — они работают. Несмотря на то что объектные базы данных могут быть более производительными в плане записи в них объектов, цена, которой обходится преобразование всех реляционных баз данных в объектные, неприемлема. Одним словом, чтобы воспользоваться объектной базой данных, той или иной компании пришлось бы преобразовать всю свою информацию из реляционной базы данных в информацию, которую можно будет разместить в объектной базе данных. У такого подхода есть много недостатков.

Во-первых, любой, кому доводилось выполнять преобразование информации из одной базы данных в информацию, которая будет располагаться в другой базе данных, знает, что это очень болезненный процесс. Во-вторых, даже если данные будут успешно преобразованы, невозможно предугадать, как смена инструментов баз данных повлияет на программный код. В-третьих, когда возникают проблемы (что случается почти всегда), сложно выяснить, связаны они с базой данных или с программным кодом. Это может превратиться в кошмар. Люди, ответственные за принятие решений в компаниях, обычно не желают так рисковать. Поэтому объектные базы данных были переведены в совершенно новые системы, написанные с использованием объектно-ориентированного кода.

Однако у нас еще имеется следующая проблема: мы хотим писать объектно-ориентированные приложения, но нам нужен доступ к унаследованным данным в реляционных базах данных. Именно здесь в дело вступает отображение из объектно-ориентированных баз данных в реляционные.

Доступ к реляционной базе данных. Все приложения баз данных имеют следующую структуру:

- клиент базы данных;
- сервер базы данных;
- база данных.

Клиент базы данных — это пользовательское приложение, обеспечивающее интерфейс определенной системы. Зачастую это приложение с графическим интерфейсом, которое дает пользователям возможность обращаться к базе данных с запросами, а также обновлять ее.

SQL

SQL означает «язык структурированных запросов». Это стандартный инструмент для общения клиентов баз данных с системами баз данных от разных поставщиков, реализующими этот стандарт.

Клиент базы данных общается с сервером базы данных с помощью SQL-операторов. На рис. 12.5 показан общий вариант модели «клиент/сервер базы данных».

В качестве примера используем Java для общения с реляционной базой данных Microsoft Access. Java задействует технологию JDBC (означает «соединение с базами данных на Java») для взаимодействия с серверами баз данных.

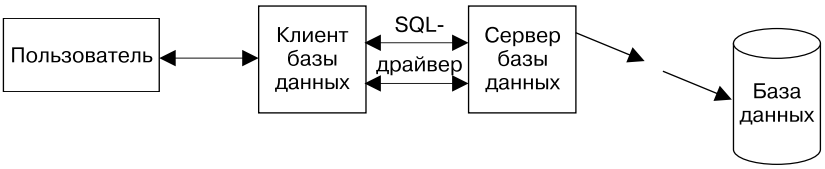

Рис. 12.5. Модель «клиент/сервер базы данных»

Часть проблемы с драйверами баз данных заключается в том, что они нередко определяются поставщиками. Это обычная проблема с драйвером любого типа. Вы, вероятно, знаете, что принтер продается в комплекте с соответствующим драйвером и после покупки вам, скорее всего, даже придется скачать обновления этого драйвера. С программными продуктами возможны похожие проблемы. Каждый поставщик предусматривает определенный протокол для взаимодействия со своим продуктом. Это решение, вероятно, будет хорошо работать и при дальнейшем использовании продукции от соответствующего поставщика. Однако если вы захотите сохранить возможность смены поставщика, то можете столкнуться с проблемами.

Компания Microsoft разработала стандарт под названием Open Database Connectivity (ODBC) — «Открытый интерфейс взаимодействия с базами данных». Как пишет Джейми Яворски в своей книге «Платформа Java 2 в действии» *(Java 2 Platform Unleashed)*, «ODBC-драйверы абстрагируют определяемые поставщиками протоколы, обеспечивая общий интерфейс программирования приложений для клиентов баз данных. Создавая свои клиенты баз данных с использованием API-интерфейса ODBC, вы сделаете так, что ваши программы смогут получить доступ к большему количеству серверов баз данных». Взгляните на рис. 12.6. На нем показано, как ODBC вписывается в эту картину.

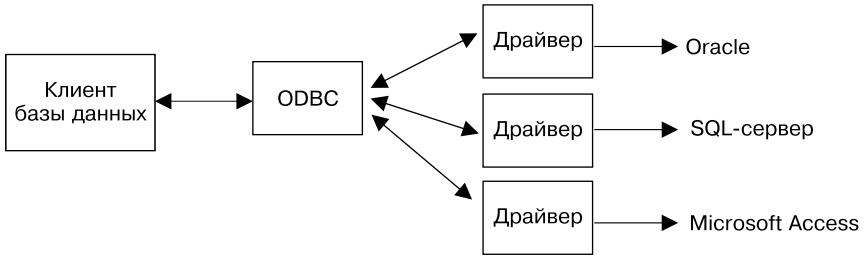

Рис. 12.6. Модель «клиент/сервер базы данных» с использованием ODBC

И снова мы видим слова *«абстрагирование»* и *«интерфейс»* в определении программного API-интерфейса. Используя ODBC, мы можем писать приложения, придерживаясь соответствующего стандарта, и нам не потребуется знать реализацию. Теоретически мы можем писать код, отвечающий стандарту ODBC, и не беспокоиться о том, что является реализацией базы данных — база данных Microsoft Access или база данных Oracle.

Как видно из рис. 12.5, клиент использует драйвер для отправки SQL-операторов серверам баз данных. Java задействует JDBC для общения с серверами баз данных.

Надо отметить, что JDBC может работать по-разному. Прежде всего, одни JDBC-драйверы позволяют подключаться к серверам баз данных напрямую. Другие же применяют ODBC для подключения к серверам баз данных, как показано на рис. 12.7.

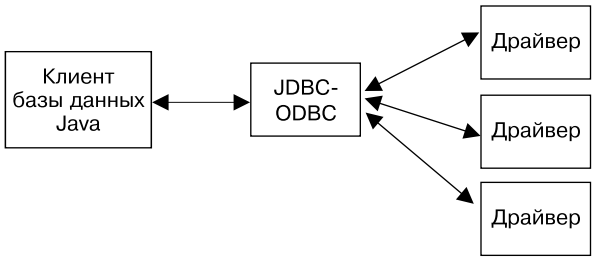

Рис. 12.7. Модель «клиент/сервер базы данных» с использованием ODBC/JDBC

В зависимости от того, как вы решите писать свои приложения, вам может потребоваться скачать разные драйверы и программные серверы. Это не относится к теме книги, которую вы сейчас читаете, поскольку здесь мы рассматриваем главным образом общие концепции. Более подробные сведения о том, как конфигурировать базы данных и подключаться к ним с использованием приложений, вы найдете в книге «Платформа Java 2 в действии» *(Java 2 Platform Unleashed)*.

Резюме

В этой главе мы разобрали концепцию постоянства объектов. Ранее мы сосредотачивались главным образом на фундаментальных объектно-ориентированных концепциях и рассматривали определенный объект как сущность, которая «живет» только на протяжении жизненного цикла приложения, его создавшего. Мы также исследовали тему объектов, которые должны продолжать существовать и после завершения жизненного цикла одного или нескольких приложений.

Например, может потребоваться, чтобы приложение восстановило объект, созданный другим приложением, или, возможно, оно будет генерировать объект для его дальнейшего использования самим этим приложением либо другими приложениями. Один из способов обеспечить постоянство объекта — сериализовать его и записать в соответствующий файл. Еще один способ состоит в использовании реляционной базы данных.

Ссылки

- *Савитч Уолтер*. Абсолютный Java (Absolute Java). 3-е изд. — Бостон: Addison-Wesley, 2008.
- *Вальтер Стивен*. ASP.NET 3.5 в действии (ASP.NET 3.5 Unleashed). — Индианаполис: Sams Publishing, 2008.
- *Скит Джон*. C#. Программирование для профессионалов: что нужно для того, чтобы освоить C# 2 и 3 (C# in Depth: What You Need to Master C# 2 and 3). — Гринвич: Manning, 2008.

- *Лейже Боб и Лейже Джеймс*. Руководство по веб-технологиям баз данных из серии книг «Web Warrior» (The Web Warrior Guide to Web Database Technologies). — Бостон: Course Technology (Cengage), 2004.
- *Дейтел и др*. C# в подлиннике. Наиболее полное руководство (C# for Experienced Programmers). — Аппер-Сэддл-Ривер: Prentice Hall, 2003.
- *Дейтел и др*. Visual Basic .NET для опытных программистов (Visual Basic .NET for Experienced Programmers). — Аппер-Сэддл-Ривер: Prentice Hall, 2003.
- *Яворски Джейми*. Платформа Java 2 в действии (Java 2 Platform Unleashed). — Индианаполис: Sams Publishing, 1999.
- *Флэнаган Дэвид и др*. Java Enterprise. Справочник (Java Enterprise in a Nutshell). — Себастопол: O'Reilly, 1999.
- *Фарли Джим*. Java и распределенные вычисления (Java Distributed Computing). — Себастопол: O'Reilly, 1998.
- Oracle: http://www.oracle.com/technetwork/java/index.html

Примеры кода, использованного в этой главе

Приведенный далее код написан на C# .NET. Эти примеры соответствуют Java-коду, продемонстрированному в текущей главе.

Пример класса Person: C# .NET

```
// Класс Person
using System;
using System.Collections;
using System.IO;
using System.Xml;
using System.Xml.Serialization;

namespace CSSerial
{
  [XmlRoot("person")]
  public class Person
  {
    private String strName;
    private int intAge;
    private int intScore;

    public Person()
    {
      this.Name = "Джон Доу";
      this.Age=25;
      this.Score=50;
    }

    public Person(String name, int age, int score)
```

```
    {
      this.Name = name;
      this.Age = age;
      this.Score = score;
    }
    [XmlAttribute("name")]
    public String Name
    {
      get
      {
        return this.strName;
      }
      set
      {
        if (value == null) return;
        this.strName = value;
      }
    }

    [XmlElement("age")]
    public int Age
    {
      get
      {
        return this.intAge;
      }
      set
      {
        this.intAge = value;
      }
    }

    [XmlIgnore()]
    public int Score
    {
      get
      {
        return intScore;
      }
      set
      {
        this.intScore = value;
      }
     }
  }
}

// Класс CSSerial
using System;
```

```
using System.Collections;
using System.IO;
using System.Xml;
using System.Xml.Serialization;

namespace CSSerial
{
  class Program
  {
    static void Main(string[] args)
    {
      Program myProgram = new Program();
    }

    public Program()
    {
      Serialize();
      DeSerialize();
    }

    public void Serialize()
    {
      Person[] myPeople = new Person[3];
      myPeople[0] = new Person("Джок Кью Паблик", 32, 95);
      myPeople[1] = new Person("Джейкоб М. Смит", 35, 67);
      myPeople[2] = new Person("Джо Л. Джонс", 65, 77);
      XmlSerializer mySerializer = new XmlSerializer(typeof(Person[]));
      TextWriter myWriter = new StreamWriter("person.xml");
      mySerializer.Serialize(myWriter, myPeople);
      myWriter.Close();
    }

    public void DeSerialize()
    {
      Person[] myRestoredPeople;
      XmlSerializer mySerializer = new XmlSerializer(typeof(Person[]));
      TextReader myReader = new StreamReader("person.xml");
      myRestoredPeople = (Person[])mySerializer.Deserialize(myReader);
      Console.WriteLine("Мои восстановленные люди:");
      foreach (Person listPerson in myRestoredPeople)
      {
        Console.WriteLine(listPerson.Name + " имеет возраст " + listPerson.Age
                                          + " лет.");
      }
      Console.WriteLine("Нажмите любую клавишу, чтобы продолжить...");
      Console.ReadKey();
    }    }
}
```

Глава 13

Объекты в веб-службах, мобильных и гибридных приложениях

Пожалуй, основная причина того, что объекты сейчас так популярны в сообществе, занимающемся разработкой программного обеспечения, связана с Интернетом. Несмотря на то что объектно-ориентированные языки существуют по сути столько же, сколько и структурные, лишь с появлением Интернета объекты получили широкое признание. В настоящее время объекты предпочтительны фактически во всех основных сетях, начиная с Интернета и заканчивая мобильными сетями, даже при передаче такого содержимого, как развлекательные и игровые медиаданные, с помощью прочно устоявшейся инфраструктуры вроде кабельных и местных телефонных линий.

Хотя объекты до недавнего времени отнюдь не имели широкого распространения (они получили его с конца 1990-х годов), объектно-ориентированный язык Smalltalk был популярен в 1980-х и 1990-х годах, а основанный на объектах язык C++ стал широко использоваться в 1990-х годах. Со временем C++ стал основным объектным языком на соответствующем рынке. Java, который изначально был нацелен именно на сети, сегодня коммерчески успешный объектно-ориентированный язык. Сейчас, с внедрением .NET и Objective-C, объектно-ориентированные языки стали частью основного потока. В этой главе рассматриваются некоторые объектные технологии, используемые в Интернете и прочих основных сетях.

Эволюция распределенных вычислений

Одна из самых удивительных особенностей жизни профессионального разработчика заключается в том, что изменения происходят непрерывно. Хотя изменения всегда волнующи и поддерживают индустрию постоянно оживленной, не обходится без издержек — необходимо поддерживать все замечательные технологии прошлого. Коротко говоря, сегодня разработчикам нужно понимать, как создать все самые современные новшества и интегрировать их с множеством унаследованных технологий. Это означает, что постоянно приходится обеспечивать поддержку ряда текущих и унаследованных систем. Только подумайте о том же коде на COBOL, написанном десятилетия назад, который все еще является жизненно важной частью инфраструктуры информационных технологий многих компаний. Реляционные базы данных — еще один хороший пример технологии, которая существует уже

десятилетия, но по-прежнему остается удивительно актуальной. История распределенных вычислений ничем не отличается от всего этого.

В общем смысле мы можем проследить развитие компьютерных технологий от распределенных вычислений до появления электронной почты. В этой книге мы сосредоточимся на процессе обмена объектами между приложениями, которые располагаются на распределенных физических платформах. Распределенные вычисления включают много технологий, в том числе следующие, которые в разной степени рассматриваются в текущей главе:

- HTML;
- EDI;
- удаленные вызовы процедур;
- CORBA;
- DCOM;
- XML;
- SOAP;
- веб-службы;
- ReST.

Основанные на объектах языки сценариев

Основное внимание в этой книге сосредоточено на объектно-ориентированных языках программирования. Такие языки, как Java, .NET и Objective-C, используются для создания полных, потенциально автономных приложений. Однако эти объектно-ориентированные языки не единственные инструменты, позволяющие программировать с использованием объектов. Ранее уже отмечалось, что C++ не является подлинным объектно-ориентированным языком, а в действительности представляет собой основанный на объектах язык программирования, поскольку в случае с ним объектно-ориентированные концепции отнюдь не обязательны. Вы можете написать не объектно-ориентированную программу на C, используя компилятор C++. Существует также класс языков, называемых языками сценариев, — в эту категорию попадают JavaScript, VBScript, ASP, JSP, PHP, Perl и Python.

ОБЩАЯ МОДЕЛЬ

Для создания веб-страниц применяются разные технологии. У всех языков программирования, языков сценариев и языков разметки есть свое место в соответствующей модели. Хотя внимание в этой книге сосредоточено в основном на объектно-ориентированных языках, важно понимать, что языки программирования являются лишь частью головоломки.

Сделаем небольшую паузу на данном этапе, чтобы рассмотреть темы, которые связаны с Интернетом и послужат основой для нашего исследования вопросов, касающихся Всемирной паутины и, соответственно, веб-служб. Сначала рассмотрим концепцию модели «клиент/сервер». На рис. 13.1 показан ее типичный вариант.

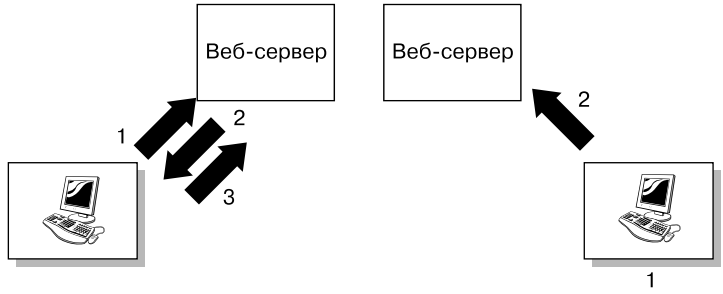

Рис. 13.1. Модель «клиент/сервер»

Важно понимать, что у модели «клиент/сервер» есть две стороны. Как видно из названия, это сторона клиента, которую во многих случаях представляет браузер, и сторона сервера, представляемая физическим веб-сервером. В данном случае хорошим упражнением послужит простой пример из сферы электронной торговли.

Допустим, вы создаете простую веб-страницу, которая будет запрашивать у пользователя следующую информацию:

- дату;
- имя;
- фамилию;
- возраст.

Эту страницу можно открыть в браузере, который считается клиентом (рис. 13.2).

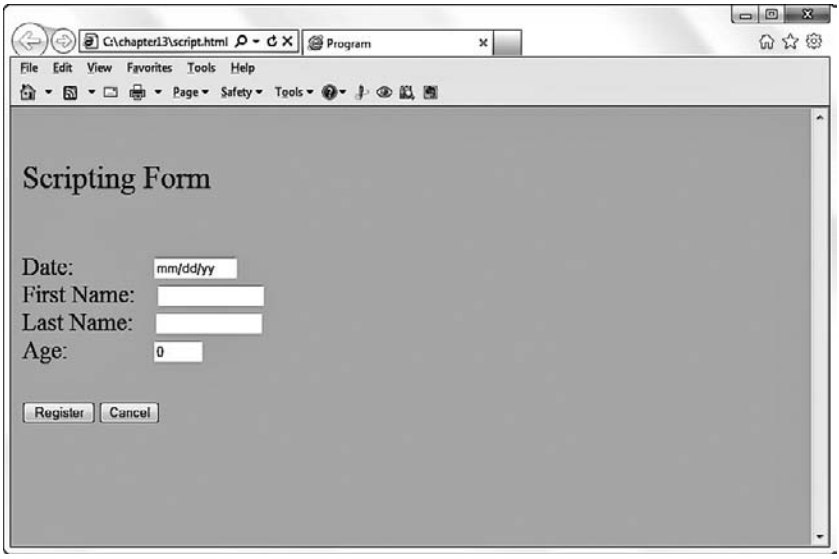

Рис. 13.2. Отображение HTML-документа

Это очень простой HTML-документ; вместе с тем он хорошо иллюстрирует концепцию валидации форм. Один из основных вопросов, которые мы должны решить при разработке системы «клиент/сервер», заключается в следующем: будем мы осуществлять валидацию на стороне клиента, на стороне сервера или же и там и там.

Допустим, нам нужно проверить, является ли допустимой дата, введенная пользователем. Нам также необходимо, чтобы возраст был указан в допустимом диапазоне — разумеется, мы не хотим, чтобы кто-нибудь ввел значение возраста в виде -5. Вопрос состоит в том, осуществлять валидацию на стороне клиента или же на стороне сервера. Посмотрим, почему это важный аспект и как он связан с объектами.

Сначала решим вопрос с полем Возраст. В большинстве бизнес-систем информация о покупателях сохраняется в базе данных, расположенной на сервере. По соображениям безопасности клиентам не предоставляется прямой доступ к этой базе.

БЕЗОПАСНОСТЬ КЛИЕНТОВ

Поскольку любой может воспользоваться браузером, было бы очень глупо предоставлять клиенту (браузеру) прямой доступ к базе данных. Таким образом, если клиенту потребуется заглянуть в базу данных или обновить ее, ему придется запросить соответствующую операцию у сервера. Это основной вопрос безопасности.

Это идеальный пример парадигмы «интерфейс/реализация», подчеркиваемой на всем протяжении этой книги. В данном случае клиент запрашивает услугу у сервера. Программная система обеспечивает интерфейс, с помощью которого клиент сможет буквально отправлять сообщения и запрашивать конкретные услуги у сервера.

Допустим, в примере, связанном с полем Возраст в HTML-документе на рис. 13.2, пользователю с именем Мэри понадобилось обновить значение своего возраста в базе данных. Открыв требуемую веб-страницу, этот пользователь вводит соответствующую информацию в форму (включая значение своего возраста в поле Возраст), а затем нажимает кнопку Регистрация. При самом простом сценарии информация, введенная в форму, будет отправлена на сервер, который затем обработает эти сведения и обновит базу данных.

Как проверяются данные, введенные в поле Дата? Если валидация не будет проводиться, то программное обеспечение, выполняющееся на сервере, обратится к полю Возраст в записи Мэри и обновит информацию. Даже если значение возраста, введенное Мэри, окажется некорректным, оно попадет в базу данных.

Если валидация будет осуществляться на сервере, то выполняющееся на нем программное обеспечение проверит, что значение в поле Возраст попадает в соответствующий диапазон. Возможно также, что сама база данных будет проводить проверку, чтобы убедиться в том, что значение возраста лежит в правильном диапазоне.

Однако у валидации на стороне сервера есть один серьезный недостаток — информация должна отправляться на сервер. Это может показаться парадоксальным, но вы можете задать следующий простой вопрос: зачем проводить валидацию чего-либо на стороне сервера, когда это можно сделать на стороне клиента?

ИЗБЫТОЧНАЯ ВАЛИДАЦИЯ

Важно отметить, что валидация должна проводиться на обеих сторонах для всех веб-приложений, так как, например, клиенты могут отправлять результаты прямо на сервер без осуществления валидации на своей стороне либо отключить клиентский сценарий и опять же отправить неверные значения.

Здесь есть несколько нюансов. Отправка данных на сервер:

- занимает больше времени;
- увеличивает сетевой трафик;
- отнимает ресурсы сервера;
- увеличивает вероятность ошибки.

По этим причинам, а также из-за других возможных проблем наша цель заключается в том, чтобы проводить *б*ольшую часть валидации на стороне клиента. Именно здесь в дело вступают языки сценариев.

Пример валидации с использованием JavaScript

JavaScript, как и большинство популярных сценарных языков, считается основанным на объектах. Как и в случае с языком C++, вы можете писать JavaScript-приложения, не отвечающие объектно-ориентированным критериям. Однако JavaScript все же обладает объектно-ориентированными возможностями. Вот что выделяет такие языки сценариев, как JavaScript и ASP.NET, в объектно-ориентированной нише. Вы можете использовать объекты в JavaScript-приложениях для расширения функционала своих веб-страниц. Эти языки в какой-то мере можно считать мостиками между традиционными парадигмами программирования и объектно-ориентированными моделями. Важно понимать, что у вас есть возможность включать объекты в свои веб-приложения, даже если вы не используете только объектно-ориентированные технологии.

Чтобы оценить мощь языков сценариев, сначала нужно понять ограничения HTML. HTML — это язык разметки, который обеспечивает функциональность и не обладает врожденными возможностями программирования. Например, с помощью HTML нельзя запрограммировать оператор `if` или цикл. Таким образом, в ранние годы HTML существовало очень немного способов валидации данных на стороне клиента. Появление сценариев изменило все это.

Благодаря функциональности, обеспечиваемой JavaScript и другими языками сценариев, разработчики веб-страниц получили возможность внедрять в них логику программирования, что позволяет осуществлять валидацию на стороне клиента. Взглянем на пример очень простого приложения для валидации с использованием HTML и JavaScript. Код этой простой веб-страницы будет выглядеть следующим образом:

```
<html>
<head>
```

```
<title>Программа для валидации</title>
<script type = "text/javascript">
function validateNumber(tForm) {
    if (tForm.result.value != 5 ) {
      this.alert ("не 5!");
    } else {
      this.alert ("Правильно. Хорошая работа!");
    }
}
</script>
</head>
<body>
<hr>
<p>
<h1>Валидация</h1>
<form name="form">
<input type="text" name="result" value="0" SIZE="2">
<input type="button" value="Валидация" name="calcButton"
  onClick="validateNumber(this.form)">
</form>
<hr>
</body>
</html>
```

В первую очередь здесь следует обратить внимание на то, что JavaScript-код вложен в HTML-код. Все это не похоже на то, как используется тот или иной язык программирования. В то время как код на языках вроде Java и C# существует как независимый программный продукт, обычно считается, что на стороне клиента JavaScript «обитает» в рамках браузера. Однако JavaScript-файлы могут существовать независимо как внешние файлы с кодом и библиотеки (например, jQuery).

JAVA В ПРОТИВОПОСТАВЛЕНИИ С JAVASCRIPT

Хотя и Java и JavaScript основаны на синтаксисе C, они не связаны друг с другом.

При отображении в клиентском браузере наша веб-страница будет выглядеть очень просто (рис. 13.3).

В этом приложении пользователь сможет ввести в поле число, а затем нажать кнопку Валидация. После этого приложение проверит, является ли введенное значение цифрой 5. Если введенное значение не окажется равным 5, то появится окно с предупреждением о том, что произошла ошибка валидации (рис. 13.4).

Если пользователь введет 5, то появится окно с предупреждением, в котором будет сказано, что именно это значение ожидалось.

Механизм этой валидации базируется на двух отдельных частях JavaScript-сценария, которыми являются:

- определения функций;
- HTML-теги.

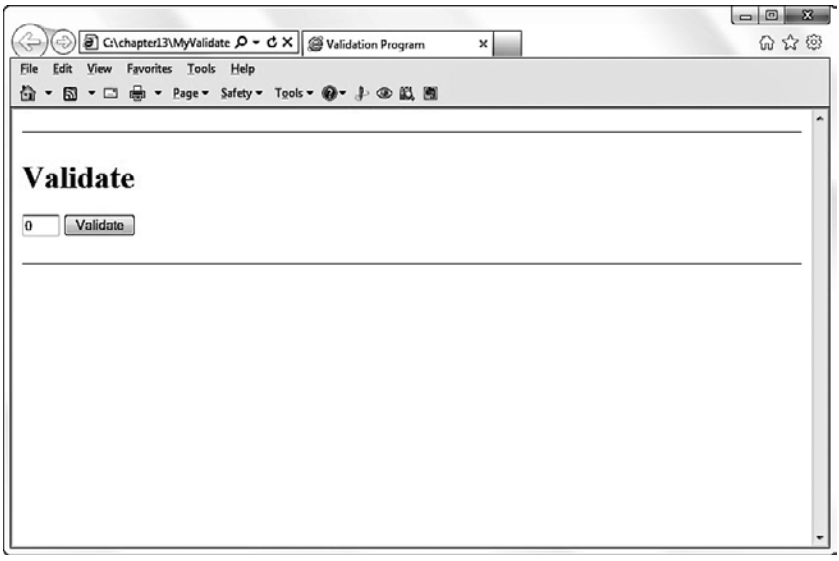

Рис. 13.3. Клиент приложения для валидации с использованием JavaScript

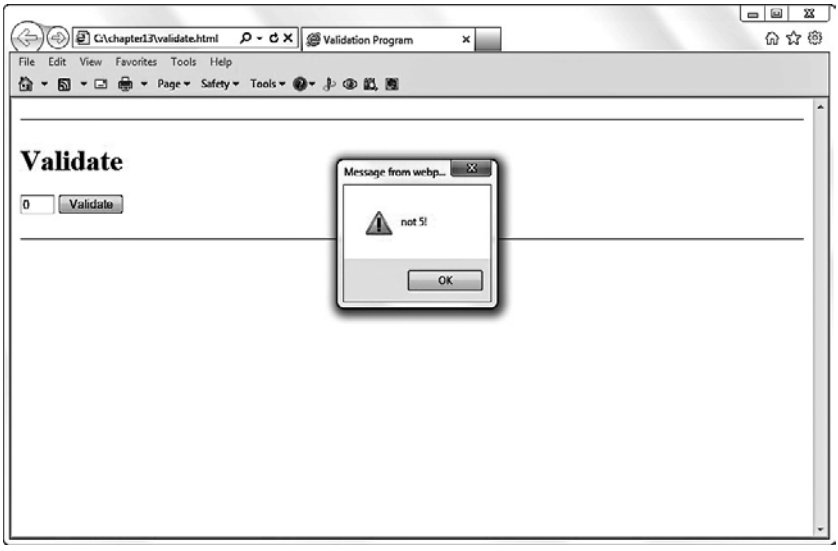

Рис. 13.4. Окно с предупреждением при валидации с использованием JavaScript

Как и при использовании регулярных языков программирования, мы можем определять функции на JavaScript. В этом примере у нас в приложении имеется одна функция `validateNumber()`:

```
<script type = "text/javascript">
function validateNumber(tForm) {
    if (tForm.result.value != 5 ) {
```

```
        this.alert ("не 5!");
      } else {
        this.alert ("Правильно. Хорошая работа!");
      }
  }
  </script>
```

JAVASCRIPT-СИНТАКСИС

Поскольку в этой книге мы больше рассматриваем концепции, вам следует обратиться к другому изданию, посвященному JavaScript, чтобы узнать об особенностях синтаксиса этого языка.

Фактический вызов функции осуществляется после нажатия кнопки Валидация. Соответствующее действие перехватывается в определении HTML-формы:

```
<input type="button" value="Validate" name="calcButton"
  onClick="validateNumber(this.form)">
```

После нажатия кнопки Валидация объект, представляющий форму, отображается в соответствии с параметрами функции `validateNumber()`.

Объекты на веб-странице

Есть много способов описания объектов в HTML-файлах для применения на веб-страницах. Объекты могут быть реализованы с помощью языков сценариев, как в примере валидации с использованием JavaScript из предыдущего раздела. Внешние объекты также могут быть включены в HTML-файлы.

Существует много примеров этих внешних объектов. Одни используются для воспроизведения медиаданных вроде музыки и фильмов. Другие могут осуществлять манипуляции с объектами, создаваемыми сторонним программным обеспечением вроде PowerPoint или Flash.

В этом разделе мы разберем, как объекты добавляются на веб-страницу.

JavaScript-объекты

Объектное программирование свойственно процессу, который происходил в JavaScript-примере, проиллюстрированном в предыдущем разделе. Это можно увидеть, взглянув на код функции `validateNumber()`. Хотя многие названия, например «компоненты», «виджеты», «элементы управления» и т. д., описывают части интерфейса пользователя, все они связаны с функциональностью объектов.

Для создания веб-страницы вы можете использовать несколько объектов, например:

- `textbox`;
- `button`;
- `form`.

Каждый из этих объектов содержит свойства и методы. К примеру, вы можете изменить значение свойства `color` объекта `button` или значение его свойства `label`.

Далее, form можно представлять себе как объект, состоящий из других объектов. Как видно из приведенной ниже строки кода, используемая нотация соответствует правилам объектно-ориентированных языков (добавление точки для отделения объекта от свойств и методов). В этой строке кода вы можете видеть, что свойство value объекта textbox (с именем result) относится и к объекту form (с именем tForm):

```
if (tForm.result.value != 5 )
```

Кроме того, alertbox как таковой является объектом. Мы можем убедиться в этом по наличию указателя this в коде:

```
this.alert ("Правильно. Хорошая работа!");
```

УКАЗАТЕЛЬ THIS

Помните, что указатель this обращен на текущий объект, которым в данном случае является form.

JavaScript поддерживает определенную иерархию объектов. На рис. 13.5 показан частичный список этой иерархии.

Рис. 13.5. Дерево JavaScript-объектов

Как и в других языках сценариев, в JavaScript предусмотрено несколько встроенных объектов. Мы можем взглянуть, к примеру, на встроенный класс Date.

Экземпляром этого класса является объект с такими методами, как `getHours()` и `getMinutes()`. Вы также можете создавать собственные специальные классы. В приведенном далее коде демонстрируется использование объекта `Date`:

```
<html>
<head>
<title>Пример объекта Date</title>
</head>
<body>
<script language="JavaScript" type = "text/javascript">

  days = new Array ( "воскресенье", "понедельник", "вторник",
                     "среда", "четверг", "пятница",
                     "суббота", "воскресенье");

  today=new Date

  document.write("Сегодня " + days[today.getDay()]);

</script>

</body>
</head>
</html>
```

Обратите внимание, что в этом примере мы создали объект `Array`, содержащий строковые значения, которые представляют собой дни недели. Мы также создали объект `today`, хранящий информацию, которая относится к текущей дате. На этой веб-странице будет отображаться текущий день недели в зависимости от даты в памяти вашего компьютера.

Элементы управления веб-страницы

В HTML-документ можно напрямую вложить объекты многих типов. Элементы управления веб-страницы состоят из большого массива заранее созданных объектов. Для внедрения этих объектов предназначен тег `<object>`. В качестве примера мы рассмотрим элемент управления `Slider`, который включим в простую веб-страницу. В приведенном далее HTML-коде показано, как использовать этот элемент управления:

```
<html>
<head>
<title>Ползунок</title>
</head>
<body>

<object classid="clsid:F08DF954-8592-11D1-B16A-00C0F0283628" id="Slider1"
width="100" height="50">
  <param name="BorderStyle" value="1" />
  <param name="MousePointer" value="0" />
  <param name="Enabled" value="1" />
  <param name="Min" value="0" />
```

```
    <param name="Max" value="10" />
  </object>

  </body>
  </html>
```

Открыв этот файл в браузере, вы увидите результат, показанный на рис. 13.6.

Обратите внимание, что это настоящий объект. Он содержит атрибуты, например `height` и `width`, а также поведения, связанные с ползунком. Значения для некоторых атрибутов задаются в параметрах, передаваемых из тега `<object>`.

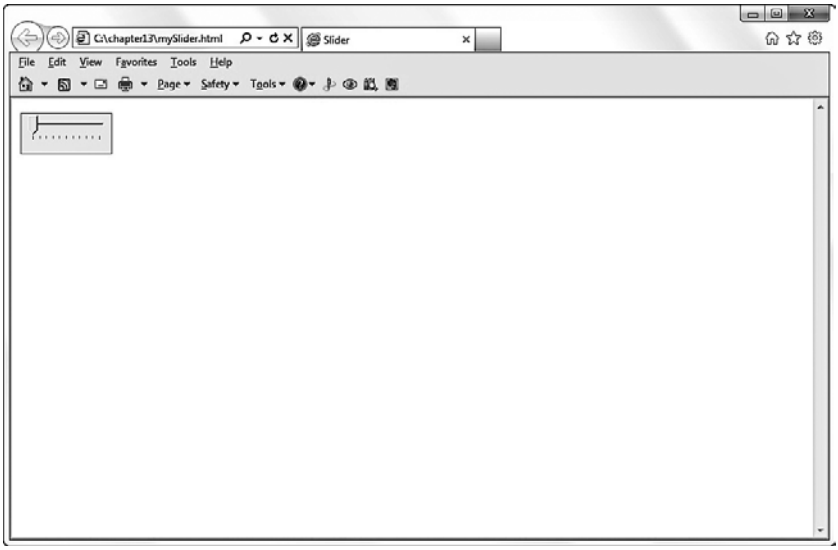

Рис. 13.6. Элемент управления на веб-странице

СОВМЕСТИМОСТЬ С БРАУЗЕРАМИ

Как и всегда, знайте, что не все объекты работают во всех браузерах или операционных системах. Эти примеры приведены с использованием браузера Internet Explorer 8 в операционной системе Windows 7.

Аудиопроигрыватели

Тег `<object>` также можно использовать для добавления на страницу разных аудиопроигрывателей и дальнейшего их запуска из браузера. В большинстве случаев то, какой из них окажется запущен, будет зависеть от проигрывателя, по умолчанию загружаемого браузером.

Например, приведенный далее HTML-код загружает и воспроизводит звуковой файл, указанный в теге `<object>`. В данном случае аудиофайл должен располагаться в соответствующей папке, хотя доступ к нему возможен через Интернет:

```
<html>
<head>
```

```
<title>Аудиопроигрыватель</title>
</head>

<body>

<object
classid="clsid:22D6F312-B0F6-11D0-94AB-0080C74C7E95">
<param name="FileName" value="fanfare.wav" />
</object>

</body>
</html>
```

Видеопроигрыватели

Видеопроигрыватели можно вкладывать точно так же, как аудиопроигрыватели. Приведенный далее код воспроизводит файл (с расширением `.wmv`) из тега `<object>`. Как и любой звуковой файл, видеофайл должен располагаться в соответствующем каталоге или по веб-адресу:

```
<html>
<head>
<title>Ползунок</title>
</head>

<body>
<object
classid="clsid:22D6F312-B0F6-11D0-94AB-0080C74C7E95">
<param name="FileName" value="AspectRatio4x3.wmv" />
</object>

</body>
</html>
```

Flash

В нашем последнем примере рассмотрим, как Flash-объект можно вложить в веб-документ с помощью того же тега `<object>`:

```
<html>
<head>
<title>Ползунок</title>
</head>
<body>

<object width="400" height="40"
classid="clsid:D27CDB6E-AE6D-11cf-96B8-444553540000"
codebase="http://download.macromedia.com
/pub/shockwave/cabs/flash/swflash.cab#4,0,0,0">
<param name="SRC" value="intro.swf">
<embed src="bookmark.swf" width="400" height="40"></embed>
```

```
</object>

</body>
</html>
```

Распределенные объекты и корпоративные вычисления

Десять или около того лет назад термин «корпоративные вычисления» стал важной частью лексикона, используемого в области информационных технологий. Сегодня значительную часть основных разработок в сфере информационных технологий составляют те, что касаются корпоративных вычислений. Но что именно означает словосочетание «корпоративные вычисления»?

Пожалуй, самое общее определение корпоративных вычислений характеризует их как распределенные вычисления. Распределенные вычисления — это, как видно из названия, вычисления, выполняемые распределенной группой компьютеров, которые работают сообща по сети. В данном случае сеть может быть проприетарной либо Глобальной.

Мощь распределенных вычислений заключается в том, что компьютеры могут разделять между собой определенную работу. В действительно распределенной среде вам даже не потребуется знать, какой именно компьютер на самом деле обслуживает ваш запрос — кроме того, возможно, будет лучше, если вы не будете этого знать. Например, если вам понадобится купить что-то через Интернет, вы зайдете на сайт соответствующей компании. Все, что вы будете знать при этом, — что вы заходите на него с использованием URL-адреса. Однако компания «подключит» вас к любой физически доступной вычислительной машине.

Почему это целесообразно? Допустим, у компании имеется один компьютер для обслуживания всех запросов. Далее представьте себе, что будет, если в работе этого компьютера произойдет сбой. Теперь предположим, что компания может распределить онлайн-операции по дюжине компьютеров. Если один из них выйдет из строя, то последствия не будут такими губительными.

Рассмотрим также случай, когда вы скачиваете файлы с сайта. Вам, вероятно, доводилось сталкиваться с ситуацией, когда на сайте загрузок предоставляются ссылки на несколько сайтов, а затем вас просят выбрать сайт, который расположен на ближайшем к вам сервере. Это означает распределение нагрузки по сети. Компьютерные сети могут самостоятельно распределять нагрузку. На рис. 13.7 приведена схема того, как может выглядеть распределенная система.

Внимание в этой книге сосредоточено на объектах и объектно-ориентированных концепциях, поэтому во многих отношениях сущности, которые нас интересуют, называются распределенными объектами. То, что объекты являются полностью независимыми, делает их идеально подходящими для распределенных приложений. В этой главе делается упор на следующее: *если приложению (клиенту) требуется служба некоего объекта (сервера), эта веб-служба может располагаться где угодно в сети, и ей на самом деле будет все равно, что именно станет ее использовать.* Рас-

смотрим некоторые технологии (прошлые и нынешние), которые имеют отношение к распределенным объектам.

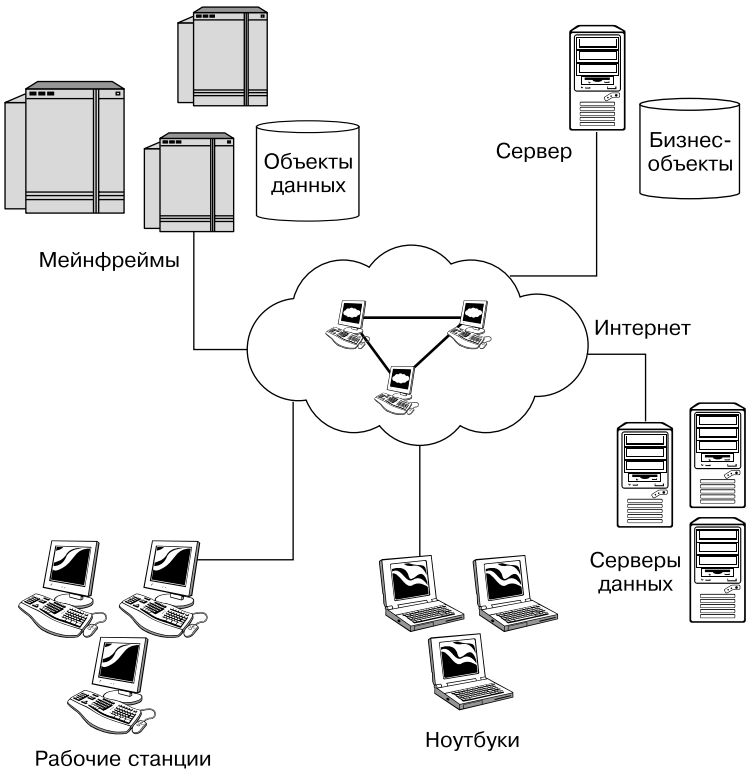

Рис. 13.7. Распределенная система

Помните, что многие из этих технологий не обязательно являются самыми современными, однако по-прежнему используются как унаследованные системы. Например, технология CORBA уже далеко не так широко применяется, как на тот момент, когда вышло первое издание этой книги в конце 1990-х годов. Сейчас, скорее всего, для реализации корзины, применяемой в интернет-магазине, будут задействованы некоторые веб-службы или она будет поддерживаться прямо в нативном формате сайта вместо использования CORBA-интерфейса. SOAP, XML и сервис-ориентированная архитектура сегодня намного более распространены. Вместе с тем небольшой исторический обзор может оказаться весьма полезным для понимания того, как эволюционировали соответствующие технологии.

Common Object Request Broker Architecture (CORBA)

Одно из основных положений этой книги заключается в том, что объекты — это полностью независимые блоки. Учитывая это, не потребуется сильно напрягать

воображение, чтобы представить себе передачу объектов по сети. Фактически мы уже использовали объекты, путешествующие по сети, во многих примерах на всем протяжении этой книги.

Вся суть корпоративных вычислений базируется на концепции распределенных объектов. Использование объектов несет в себе много преимуществ; пожалуй, самое интересное состоит в том, что система теоретически может вызывать объекты, располагающиеся где угодно в сети. Это существенная возможность, на которую опирается значительная часть современного бизнеса на основе Интернета. Еще одно важное преимущество заключается в том, что разные части системы могут быть распределены через сеть по множеству компьютеров.

Идея доступа к объектам и их вызова по сети — это очень хорошая задумка. Однако здесь есть одна явная ложка дегтя в бочке меда — нерешенная проблема переносимости. Хотя мы можем создать проприетарную распределенную сеть, тот факт, что она будет проприетарной, ведет к очевидным ограничениям. Другая проблема заключается в языке программирования. Предположим, что системе, написанной на Java, потребуется вызвать объект, написанный на C++. В самых радужных мечтах мы хотели бы создать непроприетарный, независимый от языка фреймворк для объектов в распределенной среде. Именно в этом плане технология CORBA (расшифровывается как «общая архитектура брокера объектных запросов») очень помогла сообществу разработчиков, когда Интернет впервые сделался главной силой на соответствующем рынке в середине 1990-х годов.

Основной смысл CORBA таков: используя стандартный протокол, CORBA позволяет программам от разных поставщиков взаимодействовать друг с другом. Эта способность к взаимодействию охватывает аппаратное и программное обеспечение. Таким образом, поставщики могут создавать приложения на разных аппаратных платформах и операционных системах, применяя разнообразные языки программирования, и эти приложения станут работать через сети на базе технологий от разных поставщиков.

CORBA и похожие технологии, например DCOM, можно считать промежуточным программным обеспечением для различных компьютерных приложений. Хотя CORBA представляет собой лишь один тип такого ПО (позднее вы увидите и другие реализации вроде Java-технологии RMI), концепции, стоящие за промежуточным программным обеспечением, единообразны независимо от используемого подхода. В сущности, *промежуточное программное обеспечение* оказывает услуги, благодаря которым программные процессы могут взаимодействовать друг с другом по сети. Такие системы часто называют *многоуровневыми*.

Например, на рис. 13.8 показана трехуровневая система. В данном случае уровень представления и уровень данных разделены уровнем выделения, находящимся посередине (промежуточное программное обеспечение часто ассоциируют с системами объектно-реляционного отображения). Эти процессы могут выполняться на одном или нескольких компьютерах. Именно здесь уместен термин «распределяются». Процессы (или, с точки зрения этой книги, объекты) распределяются по сети. Сеть может быть проприетарной либо Глобальной.

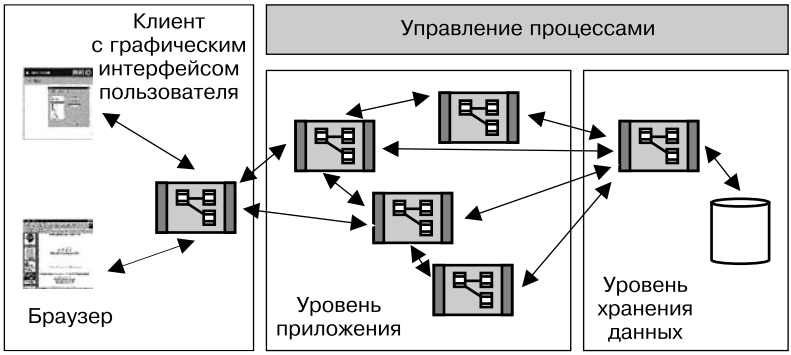

Рис. 13.8. Трехуровневая система

Вот где объекты вписываются в эту картину. Рабочая группа, занимающаяся разработкой и продвижением объектно-ориентированных технологий и стандартов (Object Management Group — OMG), отмечает: *«CORBA-приложения состоят из объектов»*. Таким образом, как вы уже знаете, объекты — это основная часть мира распределенных вычислений. Та же рабочая группа затем говорит, что эти объекты «являются отдельными блоками выполняющегося программного обеспечения, которые объединяют в себе функциональность и данные и часто (но не всегда) представляют что-то из реального мира».

Один из самых наглядных примеров такой системы — корзина, используемая в интернет-магазине. Мы можем соотнести этот пример с нашим рассмотрением создания экземпляров объектов, которое приводилось ранее. Когда вы посещаете сайт для электронной торговли, чтобы купить там товар, вам присваивается уникальная корзина, используемая в интернет-магазине. В данном случае у каждого покупателя будет объект, включающий все атрибуты и поведения объекта, который представляет корзину в интернет-магазине.

Несмотря на то что объект каждого покупателя будет содержать одинаковые атрибуты и поведения, для разных покупателей атрибутам, например `name`, `address` и т. д., будут присвоены различные значения. Объект, который представляет корзину в интернет-магазине, затем можно будет отправить куда угодно по сети. Другие объекты в системе будут представлять товары, склады и т. д.

ОБЕРТКИ

Как уже объяснялось ранее в этой книге, объекты могут широко применяться в качестве оберток. Сегодня существует немало приложений, написанных на основе унаследованных систем. Во многих случаях изменение этих унаследованных приложений либо практически нецелесообразно, либо экономически невыгодно. Один из изящных способов «подключения» унаследованных приложений к более новым распределенным системам состоит в создании объектной обертки, которая будет взаимодействовать с унаследованной системой.

Одно из преимуществ использования CORBA для реализации системы вроде нашего приложения, которое обеспечивает функционирование корзины в интернет-магазине, состоит в том, что доступ к объектам могут получить службы, созданные

с применением разных языков. Для решения этой задачи CORBA определяет интерфейс, которому должны соответствовать все языки. CORBA-концепция интерфейса хорошо согласуется с тем, о чем мы вели речь при рассмотрении проблем создания контрактов в главе 8. CORBA-интерфейс называется *языком описания интерфейсов (IDL)*. Для его функционирования необходимо, чтобы обе стороны, находящиеся на разных концах сети, то есть клиент и сервер, придерживались контракта, как указано в IDL.

В текущем исследовании будет употребляться еще один термин, значение которого мы рассмотрели ранее в книге, — маршалинг. Помните, что маршалинг — это акт, заключающийся в следующем: берется объект, выполняется его декомпозиция, чтобы придать ему формат, в котором его можно будет передать по сети, а затем этот объект восстанавливается на другом конце сети. Таким образом, если и клиент, и сервер будут придерживаться IDL, то объекты можно будет передавать по сети независимо от использованного языка программирования.

Маршрутизацию всех объектов, которые перемещаются в CORBA-системе, выполняет приложение, называемое брокером объектных запросов (Object Request Broker — ORB). Вы, возможно, уже обратили внимание, что акроним ORB является частью акронима CORBA. Именно брокер объектных запросов заставляет все двигаться в CORBA-приложении. Он заботится о маршрутизации запросов от клиентов к объектам, а также о возврате ответа в соответствующий пункт назначения.

Опять-таки мы можем увидеть, как CORBA и распределенные вычисления работают рука об руку с концепциями, разбираемыми на протяжении этой книги. Рабочая группа, занимающаяся разработкой и продвижением объектно-ориентированных технологий и стандартов, отмечает:

> «Это разделение интерфейса и реализации, которое обеспечивает IDL, является сутью CORBA».

Кроме того:

> «Клиенты получают доступ к объектам только с помощью своего заявленного интерфейса, вызывая лишь те операции, которые соответствующий объект обеспечивает с использованием своего IDL-интерфейса, лишь с теми параметрами (входными и выходными), которые включены в вызов».

Чтобы узнать, на что похож IDL, взгляните на пример из сферы электронного бизнеса, который мы разбирали в главе 8. При этом снова обратимся к UML-диаграмме, приводившейся на рис. 8.7, и создадим подмножество класса `Shop`. Если мы решим создать интерфейс `Inventory`, то сможем написать что-то вроде этого:

```
interface Inventory {
  string[] getInventory ();
  string[] buyInventory (in string product);
}
```

В данном случае у нас есть интерфейс, который определяет, как составляется список товаров и осуществляется их покупка. Затем этот интерфейс компилируется в две сущности:

- заглушки, которые выступают в качестве соединения между клиентом и брокером объектных запросов;
- скелет, выступающий в качестве соединения между брокером объектных запросов и объектом.

Эти IDL-заглушки и скелеты формируют контракт, который должны соблюдать все взаимодействующие стороны. На рис. 13.9 проиллюстрировано, как взаимодействуют разные части CORBA.

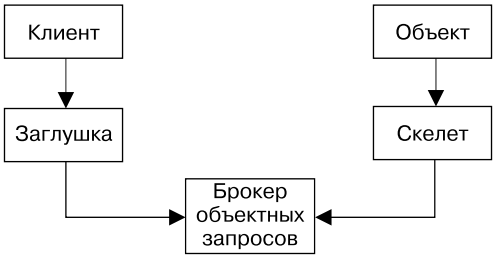

Рис. 13.9. Части CORBA

Здесь весьма любопытно то, что, когда клиенту требуется служба некоего объекта, ему не нужно ничего знать об этом объекте, в том числе и о его местоположении. Клиент просто вызывает требуемый объект (и соответствующую службу). Для клиента этот вызов представляется локальным, как если бы он вызывал объект, располагающийся в локальной системе. Этот вызов проходит через брокер объектных запросов. Если брокер объектных запросов определит, что требуемый объект является удаленным, то он выполнит маршрутизацию запроса. Если все сработает как надо, клиент не будет знать, где располагается фактический объект, который его обслуживает. На рис. 13.10 показано, как работает через сеть маршрутизация, осуществляемая брокером объектных запросов.

ПРОТОКОЛ ВЗАИМОДЕЙСТВИЯ БРОКЕРОВ ОБЪЕКТНЫХ ЗАПРОСОВ В СЕТЯХ TCP/IP

Подобно тому как HTTP является протоколом для транзакций, связанных с веб-страницами, протокол взаимодействия брокеров объектных запросов в сетях TCP/IP (Internet Inter-ORB Protocol — IIOP) представляет собой протокол для распределенных объектов, которые могут быть написаны на разных языках программирования. Протокол IIOP — это фундаментальная часть таких стандартов, как CORBA и Java-технология RMI.

В этом разделе мы рассмотрели кое-какие своеобразные и фундаментальные вопросы распределенных вычислений на основе таких пионерских технологий, как CORBA, DCOM и RMI. В последующих разделах мы продолжим наше исследование, перейдя к более поздним реализациям, — намного более распространены сегодня веб-службы с использованием, к примеру, SOAP, XML, а также сервис-ориентированная архитектура.

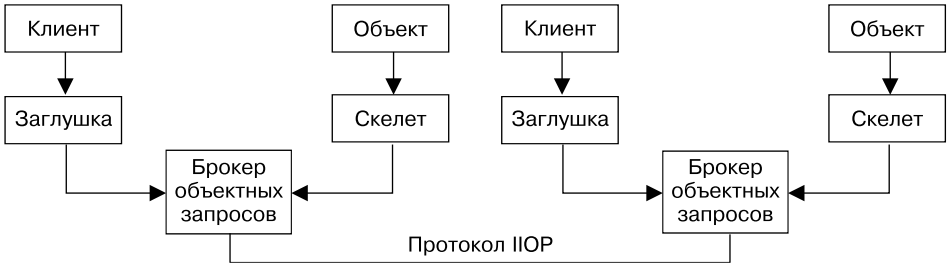

Рис. 13.10. Маршрутизация, осуществляемая брокером объектных запросов

Определение веб-служб

Веб-службы быстро эволюционировали на протяжении нескольких последних лет. Фактически, когда вышло в печать первое издание этой книги, значительная часть современных технологий пребывала в младенческом возрасте. На данном этапе мы будем использовать общее определение веб-службы, предусмотренное Консорциумом Всемирной паутины, который характеризует ее следующим образом: *«клиент и сервер, которые общаются посредством XML-сообщений с использованием стандарта SOAP (Simple Object Access Protocol — простой протокол доступа к объектам)»*.

SOAP — это коммуникационный протокол для передачи сообщений через Интернет. Теоретически SOAP не зависит от платформы и языка, а также базируется на XML. SOAP обеспечивает коммуникации между приложениями с применением протокола HTTP, поскольку пользовательские клиентские приложения обычно задействуют браузеры. SOAP расширяет функциональность HTTP для обеспечения более функциональных веб-служб.

Удаленные вызовы процедур (Remote Procedure Call — RPC) стали частью «уравнения» еще на раннем этапе эволюции распределенных вычислений. SOAP в основном направлен на осуществление удаленных вызовов процедур по протоколу HTTP с применением XML. Оставив в стороне все эти краткие описания, мы можем охарактеризовать SOAP в двух словах так: *SOAP — это основанный на XML протокол для распределенных приложений.*

УДАЛЕННЫЕ ВЫЗОВЫ ПРОЦЕДУР

Удаленные вызовы процедур — это коммуникационный процесс, который позволяет вызывать службы (объекты) с другого компьютера в совместно используемой сети. Когда системы действительно объектно-ориентированные, удаленные вызовы процедур также можно назвать удаленными вызовами методов (Remote Method Invocation — RMI). Вызов метода осуществляется без проблем в том смысле, что разработчик никогда не знает (или ему не нужно знать), является ли служба локальной или удаленной. Таким образом, фактически для вызова того или иного метода зачастую используется соответствующая подпись, скрывающая от разработчика детали реализации. Стратегии удаленных вызовов процедур также зачастую оказываются проприетарными.

Основной недостаток технологий вроде CORBA и DCOM состоит в том, что они являются проприетарными и предусматривают собственные двоичные форматы. SOAP основан на тексте, использует формат XML для сообщений и считается более

простым в применении по сравнению с CORBA и DCOM. Все это перекликается с преимуществами, обрисованными в разделе «Использование XML в процессе сериализации» главы 12.

В действительности для того чтобы беспроблемно функционировать, CORBA- и DCOM-системы должны взаимодействовать с подобными им системами. Это существенное ограничение в современной технологической среде, поскольку вы толком не знаете, что находится на другом конце сети. Таким образом, пожалуй, самое большое преимущество, которое есть у SOAP, заключается в том, что этот стандарт используется большинством крупных компаний-разработчиков программного обеспечения.

Как уже много раз подчеркивалось в этой книге, обертки — одно из основных достоинств объектной технологии. SOAP можно представлять себе как обертку, которая хоть и не является точной заменой для технологий вроде DCOM, Enterprise JavaBeans или CORBA, но «обертывает» их для более эффективной работы через Интернет. Такая возможность «обертывания» позволяет компаниям стандартизировать свои сетевые коммуникации, несмотря на то что в самих компаниях могут применяться в корне отличные технологии.

Каким бы ни было описание SOAP, важно отметить, что этот протокол, как и базовый HTML, является односторонней системой для передачи сообщений, не использующей информацию о состоянии. Из-за этой и других особенностей SOAP нельзя считать полной заменой для технологий вроде DCOM, Enterprise JavaBeans, CORBA или RMI, а можно воспринимать как технологию, которая их дополняет.

В соответствии с темой этой книги внимание в приведенном далее SOAP-примере сосредоточено на объектных концепциях, а не на конкретной SOAP-технологии, написании кода или чем-то другом.

SOAP

В соответствующем примере, приведенном в этой главе, демонстрируется «протекание» объектов через распределенную систему.

Создадим в этом примере приложение Warehouse. Оно будет использовать браузер в качестве клиента, который, в свою очередь, станет задействовать набор веб-служб для ведения деловых операций с системой Warehouse, располагающейся где-то в сети.

На рис. 13.11 приведена наглядная схема системы.

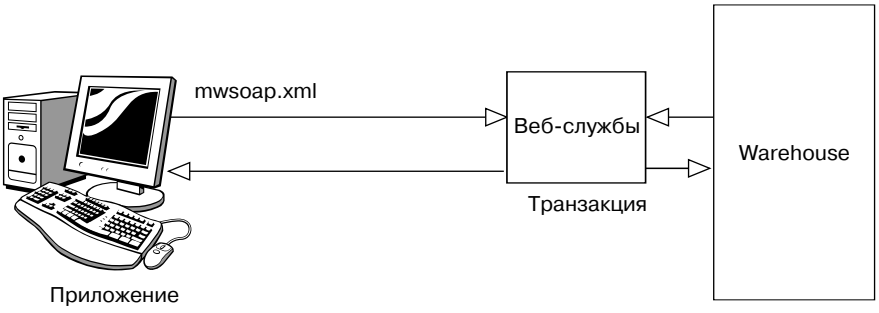

Рис. 13.11. SOAP-пример

Файл mwsoap.xml является XML-описанием структуры различных транзакций, осуществляемых веб-службами. Это описание файла Invoice.xsd показано в приведенном далее коде:

```
<?xml version="1.0" encoding="utf-8"?>
<xs:schema targetNamespace="http://ootp.org/invoice.xsd"
elementFormDefault="qualified" xmlns="http://ootp.org/invoice.xsd"
xmlns:mstns="http://ootp.org/invoice.xsd" xmlns:xs="http://www.w3.org/2001/
XMLSchema">
  <xs:element name="Invoice">
    <xs:complexType>
      <xs:sequence>
        <xs:element name="Address" minOccurs="1">
          <xs:complexType>
            <xs:sequence />
            <xs:attribute name="Street" type="xs:string" />
            <xs:attribute name="City" type="xs:string" />
            <xs:attribute name="State" type="xs:string" />
            <xs:attribute name="Zip" type="xs:int" />
            <xs:attribute name="Country" type="xs:string" />
          </xs:complexType>
        </xs:element>
        <xs:element name="Package">
          <xs:complexType>
            <xs:sequence />
            <xs:attribute name="Description" type="xs:string" />
            <xs:attribute name="Weight" type="xs:short" />
            <xs:attribute name="Priority" type="xs:boolean" />
            <xs:attribute name="Insured" type="xs:boolean" />
          </xs:complexType>
        </xs:element>
      </xs:sequence>
      <xs:attribute name="name" type="xs:string" />
    </xs:complexType>
  </xs:element>
</xs:schema>
```

Файл Invoice.xsd описывает, как структурирован Invoice и как приложения должны придерживаться его определений. Этот файл в сущности аналогичен схеме, которая используется в той или иной системе баз данных. Обратите внимание, что согласно этому файлу Invoice.xsd в состав Invoice входят Address и Package. Кроме того, в состав Address и Package входят такие атрибуты, как Description, Weight и т. д. И наконец, для этих атрибутов заданы определенные типы данных, например string, short и т. д. На рис. 13.12 наглядно показано, как выглядит соответствующее отношение.

В то время как файл Invoice.xsd описывает, *как* данные структурированы, файл mwsoap.xml представляет, *чем* они являются. Приложение, написанное на таком языке, как, например, C# .NET, VB .NET, ASP.NET или Java, будет использовать Invoice.xsd для конструирования валидных XML-файлов, которые затем будут отправлены

другим приложениям по сети. Эти приложения станут использовать один и тот же файл `Invoice.xsd`, чтобы деконструировать `mwsoap.xml` с целью его применения. Во многих отношениях вы можете считать файл `Invoice.xsd` чем-то вроде *контракта*, аналогичного по концепции контракту, рассмотренному в главе 8.

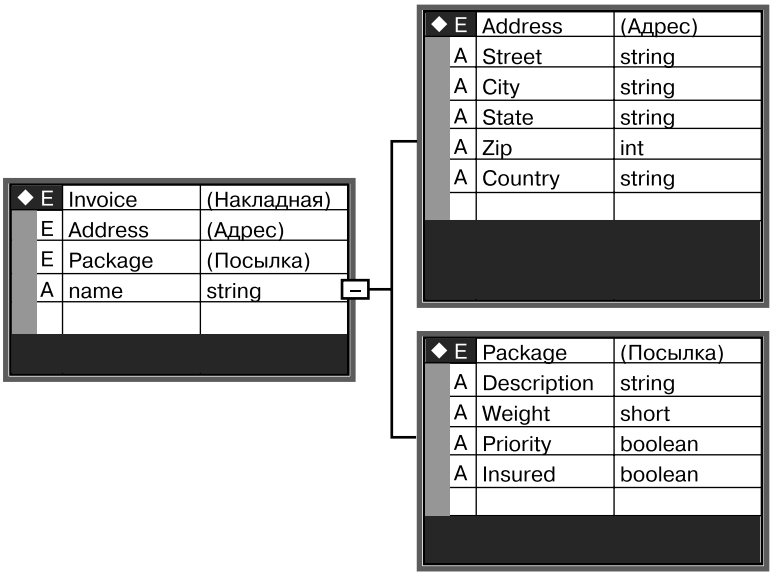

Рис. 13.12. Invoice.xsd (визуальное представление схемы)

Далее приведен файл `mwsoap.xml` с вложенными данными формата SOAP/XML:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:envelope xmlns:soap="http://www.w3.org/2001/06/soap-envelope">
  <soap:Header>
    <mySOAPHeader:transaction xmlns:mySOAPHeader="soap-transaction"
soap:mustUnderstand="true">
      <headerId>8675309</headerId>
    </mySOAPHeader:transaction>
  </soap:Header>
  <soap:Body>
    <mySOAPBody xmlns="http://ootp.org/Invoice.xsd">
      <invoice name="Дженни Смит">
        <address street="Уок-Лэйн, 475"
                 city="Самвересвилль"
                 state="Небраска"
                 zip="23654"
                 country="США"/>
        <package description="22-дюймовый плазменный монитор"
                 weight="22"
                 priority="false"
                 insured="true" />
```

```
        </invoice>
      </mySOAPBody>
    </soap:Body>
  </soap:envelope>
```

Код веб-служб

Нам осталось рассмотреть единственную часть модели — программные приложения как таковые. В приведенном далее коде на C# .NET представлены три класса, которые соответствуют `Invoice`, `Address` и `Package`.

Важно отметить, что приложения могут быть написаны на любом языке. В этом и заключается вся прелесть SOAP/XML-подхода. Каждое приложение должно быть способно осуществлять разбор соответствующего XML-файла, и это по сути единственное требование (рис. 13.13). Как приложение станет использовать извлеченные данные, будет полностью зависеть от него самого.

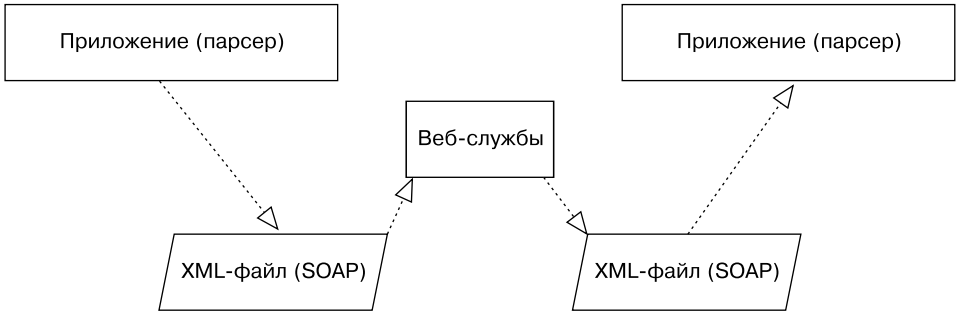

Рис. 13.13. Разбор SOAP/XML-файла

Из этого подхода можно понять, что неважно, каким именно окажется язык или, если на то пошло, платформа. Теоретически операция разбора может проводиться с использованием любого языка, что по сути и требуется.

Нам как разработчикам целесообразно напрямую взглянуть на код. Написанный на C# .NET код представлен ниже для того, чтобы показать реализацию системы, изображенной на рис. 13.12.

Invoice.cs. Приведенный далее код является C# .NET-реализацией класса `Invoice`, продемонстрированного на рис. 13.12:

```
using System;
using System.Data;
using System.Configuration;
using System.Xml;
using System.Xml.Serialization;
namespace WebServices
{
  [XmlRoot("invoice")]
  public class Invoice
  {
    public Invoice(String name, Address address, ShippingPackage package)
```

```
      {
        this.Name = name;
        this.Address = address;
        this.Package = package;
      }

      private String strName;
      [XmlAttribute("name")]
      public String Name
      {
        get { return strName; }
        set { strName = value; }
      }

      private Address objAddress;
      [XmlElement("address")]
      public Address Address
      {
        get { return objAddress; }
        set { objAddress = value; }
      }

      private ShippingPackage objPackage;
      [XmlElement("package")]
      public ShippingPackage Package
      {
        get { return objPackage; }
        set { objPackage = value; }
      }
    }
  }
```

Representational State Transfer (ReST)

По своей природе программисты нуждаются в простых, прямых подходах, в то время как менеджеры предпочитают структурированные практики. Это видно по использованию языков программирования, например Java и .NET, в сопоставлении с применением языков вроде Perl и PHP, между XML и JSON и т. д. Веб-службы тоже являются объектом нашего исследования. Многие разработчики считают такие технологии, как CORBA, удаленные вызовы процедур и SOAP, достаточно сложными. Поэтому более простой подход, называемый Representational State Transfer (переводится как «передача репрезентативного состояния»), или ReST, становится довольно популярным.

Например, SOAP-реализация может потребовать применения разнообразных инструментов (и даже иметь проблемы при взаимодействии с реализациями на основе предшествующих SOAP-стандартов), в то время как ReST требует только поддержки HTTP и может работать с разными технологиями, включая SOAP.

ReST — это протокол без сохранения состояния, который в сущности опирается на HTTP. Поскольку HTTP — основа самого Интернета, во многих отношениях

можно сказать, что архитектура Сети базируется на ReST, часто называемом архитектурой ReST.

Существенной особенностью ReST является то, что эта технология задействует HTTP-запросы для создания, обновления, считывания и удаления данных. Вследствие этого ReST может использоваться вместо других технологий вроде удаленных вызовов процедур или веб-служб, как, например, SOAP, поскольку обеспечивает всю основную требуемую функциональность. Одним словом, ReST представляет собой облегченный подход к веб-службам. Вместе с тем он никоим образом не является таким же стандартом, как XML (в одном из своих описаний ReST называется стилем). На рис. 13.14 показан поток данных, типичный для HTTP-запросов/ответов в ReSTful-реализации.

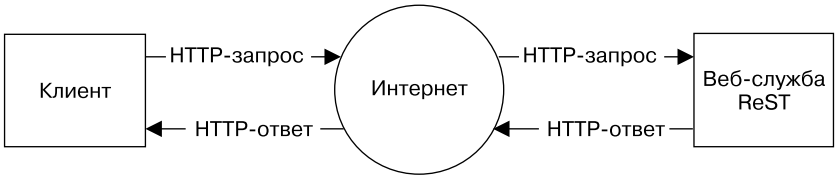

Рис. 13.14. Веб-служба ReSTful

Хотя другие стратегии вроде SOAP более замысловаты в плане того, что они способны предоставлять, очень вероятно, что ReSTful-приложение возвратит простой текст или XML-файл. Технология SOAP, в отличие от ReST, тесно связана с XML. Кроме того, ReST-запросы главным образом являются простыми текстовыми параметрами и редко предоставляют что-либо сложное. Еще одна любопытная особенность ReST заключается в том, что эта технология не объектно-ориентированная по своей природе, поскольку в действительности не предоставляет объектов, хотя они могут создаваться приложениями с использованием парсеров.

Резюме

В этой главе мы рассмотрели некоторые доступные технологии, позволяющие использовать объекты в сочетании с веб-приложениями. Важно различать объекты, вложенные в веб-страницы (например, JavaScript-объекты), и объекты, используемые в распределенных системах, а также то, как они соотносятся друг с другом.

Распределенные объекты быстро эволюционировали на протяжении нескольких последних лет. Сейчас в нише распределенных объектов доступно много вариантов; вместе с тем SOAP в сочетании с XML сделали проектирование распределенных систем намного более стандартным.

Ссылки

- *Савитч Уолтер*. Абсолютный Java (Absolute Java). 3-е изд. — Бостон: Addison-Wesley, 2008.

- *Вальтер Стивен*. ASP.NET 3.5 в действии (ASP.NET 3.5 Unleashed). — Индианаполис: Sams Publishing, 2008.
- *Скит Джон*. C#. Программирование для профессионалов: что нужно для того, чтобы освоить C# 2 и 3 (C# in Depth: What You Need to Master C# 2 and 3). — Гринвич: Manning, 2008.
- *Дейтел и др*. C# в подлиннике. Наиболее полное руководство (C# for Experienced Programmers). — Аппер-Сэддл-Ривер: Prentice Hall, 2003.
- *Дейтел и др*. Visual Basic .NET для опытных программистов (Visual Basic .NET for Experienced Programmers). — Аппер-Сэддл-Ривер: Prentice Hall, 2003.
- *Коналлен Джим*. Разработка веб-приложений с использованием UML (Building Web Applications with UML). — Бостон: Addison-Wesley, 2000.
- *Яворски Джейми*. Платформа Java 2 в действии (Java 2 Platform Unleashed). — Индианаполис: Sams Publishing, 1999.

Глава 14
Объекты и клиент-серверные приложения

В главе 13 была рассмотрена концепция распределенных объектов в их связи с веб-службами. В той главе Интернет был основной *магистралью*, по которой перемещались объекты. В текущей главе мы немного сузим рамки и исследуем тему передачи объектов по сети «клиент/сервер».

Многие концепции в этой главе тесно связаны с рассмотренными в главе 13 (поскольку и те и другие подразумевают передачу данных по сети). Я считаю, что будет очень полезно углубиться во внутреннее устройство модели «клиент/сервер» хотя бы ради образовательного опыта в разработке рабочей реализации. Коротко говоря, в этой главе мы напишем функционально полное клиент-серверное приложение.

Несмотря на то что построение маленькой образовательной модели веб-службы — более сложный процесс, мы сможем справиться с поставленной задачей на одном компьютере. Хотя объекты в распределенной сети не обязательно придерживаются определенного пути, путешествие объекта «клиент/сервер» скорее является двухточечным — по крайней мере, в концептуальном смысле.

Подходы «клиент/сервер»

Как мы уже видели в предыдущих главах, XML значительно повлиял на технологии разработки, которые используются в настоящее время. Например, распределенная объектная модель может быть построена как на основе проприетарной системы, так и с использованием непроприетарного подхода на базе технологий вроде SOAP/XML.

То же самое можно сказать о модели «клиент/сервер». Приложение может быть создано исключительно на основе проприетарной системы либо конструкции с использованием XML. В этой главе рассматриваются обе модели. Мы воспользуемся Java для описания проприетарного подхода, который будет работать только в Java-среде, хотя этот язык также может быть задействован как непроприетарное решение. Затем для иллюстрирования подхода на основе XML будет использован C# .NET.

Проприетарный подход

В этом примере Java будет применяться для демонстрации того, как устанавливается прямое двухточечное соединение по сети. Для этого я обращусь к примеру, который использую в качестве инструмента обучения уже много лет. Он заключается

в отправке объекта от клиента к серверу с последующим выводом части информации, содержащейся в этом объекте.

Основной поток для этого подхода проиллюстрирован на рис. 14.1.

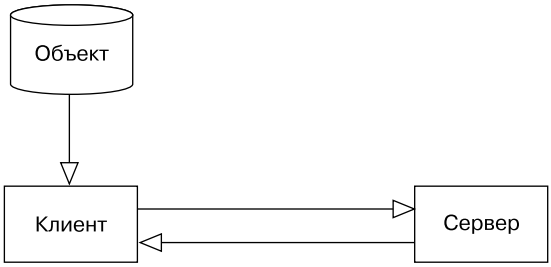

Рис. 14.1. Основной поток «клиент/сервер»

В этой конструкции клиент создает объект, который затем отправляется серверу. Сервер генерирует ссылку на объект для доступа к нему. После этого сервер может обновить атрибуты объекта и отправить его обратно клиенту.

Сериализованный объектный код

Мы начнем с создания простого класса `TextMessage` с атрибутами `name` и `message`. Этот класс также будет содержать конструктор вместе с геттерами и сеттерами. Полный класс `TextMessage` представлен в следующем коде:

```
import java.io.*;
import java.util.*;

public class TextMessage implements Serializable {

  public String name;
  public String message;

  // Конструктор класса TextMessage
  TextMessage(String n) {
    message = " ";
    name= n;
  }

  // Функция-геттер по отношению к объектам
  public String getName() {
    return name;
  }

  // Функция-геттер по отношению к объектам
  public String getTextMessage() {
    return message;
  }

  // Функция-сеттер по отношению к объектам
```

```
    public void setTextMessage(String inTextMessage) {
      message = inTextMessage;
    }
  }
```

Это довольно простой класс. Конструктор инициализирует атрибут `name` с помощью параметра и задает для `message` пустое значение. Здесь следует обратить внимание на сериализацию класса в проприетарный двоичный формат Java.

Клиентский код

Клиентский код задействует класс `TextMessage` для того, чтобы создать объект и отправить его в путешествие по направлению к серверу и обратно. Клиент должен выполнить следующие задачи.

1. Извлечь пользовательские данные.
2. Создать объект.
3. Задать значения для атрибутов.
4. Создать подключение к сокету.
5. Создать выходные потоки.
6. Записать объект.
7. Закрыть потоки.

Код для соответствующего клиента показан ниже. Комментарии поясняют большую часть этого кода.

```
import java.io.*;
import java.net.*;

/*
 * Клиент для TextMessage
 */
public class Client {

  public static void main(String[] arg) {
    try {

      String message = " ";
      String name = " ";

      System.out.print("Пожалуйста, введите имя: ");
      name = getString();

      // Создать объект TextMessage
      TextMessage myTextMessage = new TextMessage(name);

      System.out.print("сообщение: ");

      message = getString();
```

```
        // Использовать сеттер для задания TextMessage
        myTextMessage.setTextMessage(message);

        // Создать подключение к сокету
        Socket socketToServer = new Socket("127.0.0.1", 11111);

        // Создать ObjectOutputStream
        ObjectOutputStream myOutputStream = new
        ObjectOutputStream(socketToServer.getOutputStream());

        // Записать объект myTextMessage object в OutputStream
        myOutputStream.writeObject(myTextMessage);

        // Закрыть потоки
        myOutputStream.close();

      } catch (Exception e) {System.out.println(e);}
    }

    public static String getString() throws Exception {

      // открыть клавиатуру для ввода (с присвоением имени 'stdin')
      BufferedReader stdin =
        new BufferedReader(new InputStreamReader(System.in), 1);

      String s1 = stdin.readLine();

      return (s1);

    }
  }
```

Наиболее важные моменты, которые следует отметить в этом коде, «вращаются» вокруг сетевых подключений. Приведенная далее строка определяет, где клиент будет подключаться к серверу в этом примере:

```
Socket socketToServer = new Socket("127.0.0.1", 11111);
```

При создании сокета передаются два параметра, представляющие IP-адрес и виртуальный сокет, к которому клиент будет пытаться подключиться.

IP-адрес 127.0.0.1 является *шлейфовым*, а это означает, что клиент попытается подключиться к локальному серверу. Коротко говоря, клиент и сервер будут выполняться на одном и том же компьютере. Единственное очевидное условие заключается в том, что сервер должен быть запущен первым.

Такой шлейфовый IP-адрес очень удобно использовать при тестировании приложений. Вместо того чтобы требовать подключения к сети, основополагающая логика приложения может быть протестирована локально, что значительно упрощает первоначальное тестирование. Позднее можно будет провести более общее тестирование с использованием *реального* IP-адреса.

Помимо IP-адреса, в списке параметров должен быть указан виртуальный порт. В данной ситуации было выбрано произвольное значение 11111. Единственное условие для применения этого значения звучит так: сервер, к которому попытается подключиться клиент, должен прослушивать именно этот порт.

После того как клиент надлежащим образом подключится к серверу, а объект будет отправлен и извлечен, выполнение клиентского приложения завершится.

Осталось отметить здесь метод в конце класса, который выполняет такую задачу, как извлечение строки, печатаемой на клавиатуре. Эти вводимые пользователем данные сродни текстовому сообщению, которое вы набираете на своем сотовом телефоне.

Серверный код

Серверному коду на другом конце сети потребуется выполнить следующие задачи.

1. Создать ссылку на объект.
2. Прослушать виртуальный порт 11111.
3. Ждать подключения клиента.
4. Создать входные/выходные потоки.
5. Выполнить считывание объекта TextMessage.
6. Вывести сообщение.

Код для сервера приведен далее:

```
import java.io.*;
import java.net.*;

/*
 * Сервер для TextMessage
 */
public class Server {

  public static void main(String[] arg) {

    // Создать ссылку на объект, который поступит от клиента
    TextMessage myTextMessage = null;

    try {

      // Начать прослушивание сервером порта 11111
      ServerSocket myServerSocket = new ServerSocket(11111);

      System.out.println("Готов\n");

      // Ждать здесь, пока клиент не попытается подключиться
      Socket incoming = myServerSocket.accept();

      // Создать ObjectInputStream
      ObjectInputStream myInputStream = new
```

```
        ObjectInputStream(incoming.getInputStream());

        // Произвести считывание объекта из сокета с клиентом
        myTextMessage = (TextMessage)myInputStream.readObject();

        System.out.println(myTextMessage.getName() + " : "
          + myTextMessage.getTextMessage()+ "\n");

        // Закрыть потоки
        myInputStream.close();

      } catch(Exception e) {
        System.out.println(e);
      }
    }
  }
```

Как и в случае с клиентом, циклы в коде отсутствуют. Довольно просто использовать цикл для того, чтобы сервер мог постоянно прослушивать требуемый порт, однако эта функциональность не важна в рассматриваемой здесь теме.

Кроме того, сервер может обновить проект и отправить его клиенту. Клиент, к примеру, также мог бы создать входной поток и выполнить считывание объекта обратно с сервера — точно так же, как сервер может создать выходной поток и отправить объект назад клиенту.

Выполнение примера «клиент/сервер» на основе проприетарного подхода

Для упрощения пример «клиент/сервер» будет выполняться из базовой командной строки, благодаря чему нам не потребуется создавать графический интерфейс пользователя или запускать его из интегрированной среды разработки. В следующем разделе мы создадим модули, которые будут запускаться из графического интерфейса пользователя и интегрированной среды разработки.

Первый этап процесса — запуск сервера. Затем, из второго окна командной строки, будет запущен клиент. Сервер выведет сообщение, говорящее о том, что он готов и ждет. После запуска клиент запросит имя и сообщение, которое должен будет ввести пользователь.

После того как клиентские данные будут введены и отправлены, сервер отобразит сообщение, полученное от клиента. На рис. 14.2 показан серверный сеанс, а на рис. 14.3 — клиентский сеанс. Опять-таки как сервер, так и клиент могут предусматривать циклы, которые позволят совершить несколько заходов. Этот пример был сделан максимально простым для того, чтобы проиллюстрировать соответствующую технологию.

Простой пользовательский интерфейс клиента запрашивает имя пользователя, а также сообщение, которое этот пользователь желает отправить. В реальной системе, позволяющей передавать текстовые сообщения, вроде сотового телефона сервер задействовал бы адрес, введенный пользователем (по сути телефонный номер), для того, чтобы переслать сообщение второму пользователю, а не просто вывести это сообщение.

Рис. 14.2. Выполнение сервера

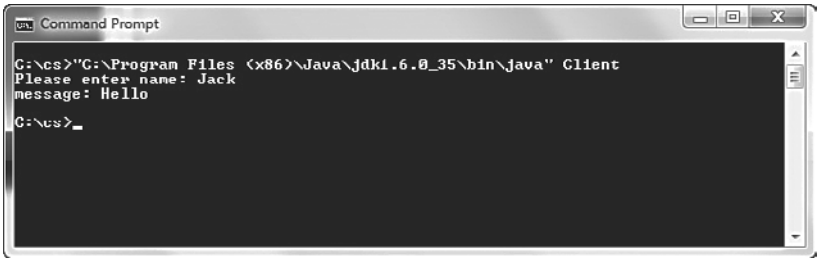

Рис. 14.3. Выполнение клиента

ПАКЕТНЫЕ ФАЙЛЫ

Для облегчения тестирования простых, но функциональных образовательных примеров такого рода мне нравится создавать старомодные пакетные файлы, чтобы сделать ввод путей к Java-классам более точным. Это не только частично устраняет необходимость постоянно печатать длинные и сложные команды в командной строке, но и позволяет протестировать несколько разных версий Java-кода, которые могут располагаться на одном компьютере. Такая методика хорошо работает в сочетании со многими другими методологиями тестирования. Например, вы можете запустить сервер, поместив приведенный далее код в пакетный файл server.bat, а затем введя server в командной строке:

```
"C:\Program Files (x86)\Java\jdk1.6.0_35\bin\java" Server
```

Этот подход также работает, когда речь идет о компиляции:

```
"C:\Program Files (x86)\Java\jdk1.6.0_35\bin\javac" Client.java
"C:\Program Files (x86)\Java\jdk1.6.0_35\bin\javac" Server.java
```

Непроприетарный подход

В предыдущем примере использовался проприетарный подход. Чтобы подход был непроприетарным, мы можем прибегнуть к технологии XML, как делали это, когда речь шла о постоянстве данных и распределенных объектах.

Использование XML-подхода позволит нам перемещать объекты туда и обратно между приложениями, написанными на разных языках, и теоретически между разными платформами. Нашу модель можно обновить, чтобы она отражала это, как показано на рис. 14.4.

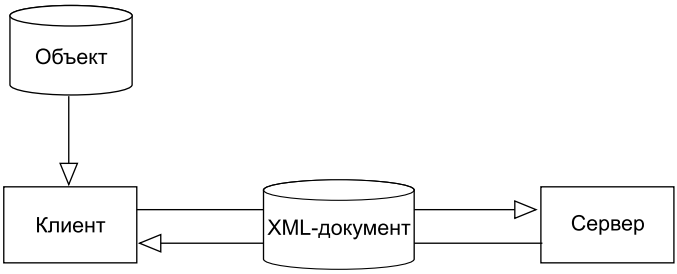

Рис. 14.4. XML-подход к коммуникациям «клиент/сервер»

Несмотря на то что многие базовые концепции здесь остаются теми же, в данном случае принципиальный способ декомпозиции и восстановления объектов предполагает переход с проприетарного, двоичного формата на непроприетарный XML-формат на основе текста.

Ради некоторого разнообразия обратимся к примеру на базе класса `CheckingAccount`.

Код определения объектов

При создании XML-версии класса `CheckingAccount` мы, проинспектировав код, сможем сразу же увидеть, что XML-определение объекта вкладывается прямо в класс (описание этого подхода можно найти в главе 11). Далее приведен код на C# .NET для класса `CheckingAccount`:

```
using System;
using System.Collections;
using System.IO;
using System.Xml;
using System.Xml.Serialization;

namespace Server
{
  [XmlRoot("account")]
  public class CheckingAccount
  {
    private String _Name;
    private int _AccountNumber;

    [XmlElement("name")]
    public String Name
    {
      get { return _Name; }
      set { _Name = value; }
    }

    [XmlElement("account_num")]
    public int AccountNumber
    {
```

```
        get { return _AccountNumber; }
        set { _AccountNumber = value; }
      }

      public CheckingAccount()
      {
        _Name = "Джон Доу";
        _AccountNumber = 54321;
        Console.WriteLine("Открытие текущего счета!");
      }
    }
  }
```

Весьма интересная особенность приведенного определения класса заключается в том, что, хотя этот класс содержит необходимые атрибуты и методы, атрибуты при этом включают свойства, соответствующие XML-определениям атрибутов.

Коротко говоря, в .NET-примерах (C# и VB) классы строятся вокруг XML-определений. Это также можно сделать с помощью Java. Фактически, задействуя XML-подход, мы можем попеременно использовать любой язык или платформу, которая нам потребуется. В этом и заключается вся прелесть непроприетарного подхода.

Обратите также внимание, что в этих C# .NET-примерах мы создаем пространство имен для наших проектов.

Клиентский код

В данном примере клиенту потребуется выполнить следующие задачи.

1. Создать объект `CheckingAccount`.
2. Создать сокет.
3. Сериализовать объект в XML-формат.
4. Создать поток.
5. Сериализовать объект в поток.
6. Закрыть ресурсы.
7. Закрыть потоки.

В большинстве случаев комментарии позволяют понять процесс выполнения программы. Далее приведен клиентский код на C# .NET:

```
using System;
using System.Collections;
using System.IO;
using System.Xml;
using System.Xml.Serialization;
using System.Net.Sockets;
using System.Net;
using System.Text;

namespace Client
{
```

```
class Client
{
  public static void Connect()
  {
    CheckingAccount myAccount = new CheckingAccount();
    try
    {
      // Создать наш TCP-сокет
      TcpClient client = new TcpClient("127.0.0.1", 11111);

      // Подготовиться к сериализации нашего объекта CheckingAccount
      // в XML-формат
      XmlSerializer myXmlFactory =
      new XmlSerializer(typeof(CheckingAccount));

      // Создать наш TCP-поток
      NetworkStream stream = client.GetStream();

      // Сериализовать наш объект в TCP-поток
      myXmlFactory.Serialize(stream, myAccount);

      // Закрыть все наши ресурсы
      stream.Close();
      client.Close();
    }
    catch (Exception ex)
    {
      Console.WriteLine("Исключение: {0}", ex);
    }

    Console.WriteLine("Нажмите любую клавишу, чтобы продолжить...");
    Console.ReadKey();
  }
 }
}
```

Серверный код

В данном случае мы воспользуемся циклом (на самом деле будет задействована пара циклов) для реализации этой версии сервера. Сервер будет отвечать за то, чтобы осуществить следующие функции.

1. Создать ссылки на объект `CheckingAccount`.
2. Подключиться к сокету и вести прослушивание.
3. Сконфигурировать входной поток.
4. Создать поток.
5. Произвести считывание байтов из потока.
6. Сериализовать объект в поток.
7. Все закрыть.

Далее приведен код на C# .NET для сервера:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Net.Sockets;
using System.Net;
using System.Xml;
using System.Xml.Serialization;
using System.IO;
using System.Runtime.Serialization;

namespace Server
{
  class Server
  {
    public Server()
    {
      TcpListener server = null;
      TcpClient client = null;
      try
      {
        // Создать наш прослушиватель сокета и запустить его
        server = new TcpListener(IPAddress.Parse("127.0.0.1"),
                                 11111);
        server.Start();

        // Сконфигурировать наш входной буфер
        Byte[] bytes = new Byte[256];

        // Совершать цикл бесконечно
        while (true)
        {
          // Начать прием входящих передач в блочном режиме
          client = server.AcceptTcpClient();
          Console.WriteLine("Подключение осуществлено!");

          // Открыть наш поток
          NetworkStream stream = client.GetStream();

          // Произвести считывание всех данных из потока
          int i;
          while ((i = stream.Read(bytes, 0, bytes.Length)) != 0)
          {
            // Подготовить формат, который сможет прочитать сериализатор
            MemoryStream ms = new MemoryStream(bytes);
            // Подготовить сериализатор
            XmlSerializer myXmlFactory =
              new XmlSerializer(typeof(CheckingAccount));
            // Создать наш CheckingAccount с использованием потока
```

```
            myRestoredAccount =
              (CheckingAccount)myXmlFactory.Deserialize(ms);
            // Теперь показать, что объект действительно был создан
            Console.WriteLine("Имя: {0}, Номер счета: {1}.",
                              myRestoredAccount.Name,
                              myRestoredAccount.AccountNumber);
            // Выбросить исключение для выхода из цикла
            throw new Exception("игнорировать");
          }
        }
      }
      catch (Exception ex)
      {
        if (!ex.Message.Equals("игнорировать"))
        { Console.WriteLine("Исключение: {0}", ex); }
      }
      finally
      {
        // Закрыть наши ресурсы
        client.Close();
        server.Stop();
      }
      Console.WriteLine("Нажмите любую клавишу, чтобы продолжить...");
      Console.ReadKey();
    }
  }
}
```

Выполнение примера «клиент/сервер» на основе непроприетарного подхода

Для выполнения этого примера вы можете создать проект с помощью Visual Studio и запустить код на C#. NET, воспользовавшись простым приложением, как показано далее:

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Threading;

namespace Server
{
  class Program
  {
    static void Main(string[] args)
    {
      Server server = new Server();
    }
  }
}
```

Резюме

В этой главе мы рассмотрели концепцию соединения «клиент/сервер». Мы применили два подхода. Первый заключался в использовании Java с целью создания проприетарной, двоичной системы для перемещения объектов с помощью сетевых подключений. Второй подход был непроприетарным и состоял в использовании .NET (как C#, так и VB). При этом непроприетарном сценарии на основе XML можно было также воспользоваться Java.

Важность этой главы, как и глав 11 и 13, заключается в том, что открытый формат, например XML, может быть использован наряду с проприетарными решениями для перемещения объектов по разным сетям, будь то сеть с двухточечным соединением или же распределенная сеть.

Ссылки

- *Савитч Уолтер*. Абсолютный Java (Absolute Java). 3-е изд. — Бостон: Addison-Wesley, 2008.
- *Вальтер Стивен*. ASP.NET 3.5 в действии (ASP.NET 3.5 Unleashed). — Индианаполис: Sams Publishing, 2008.
- *Скит Джон*. C#. Программирование для профессионалов: что нужно для того, чтобы освоить C# 2 и 3 (C# in Depth: What You Need to Master C# 2 and 3). — Гринвич: Manning, 2008.
- *Дейтел и др*. C# в подлиннике. Наиболее полное руководство (C# for Experienced Programmers). — Аппер-Сэддл-Ривер: Prentice Hall, 2003.
- *Дейтел и др*. Visual Basic .NET для опытных программистов (Visual Basic .NET for Experienced Programmers). — Аппер-Сэддл-Ривер: Prentice Hall, 2003.
- *Яворски Джейми*. Платформа Java 2 в действии (Java 2 Platform Unleashed). — Индианаполис: Sams Publishing, 1999.
- *Флэнаган Дэвид и др*. Java Enterprise. Справочник (Java Enterprise in a Nutshell). — Себастопол: O'Reilly, 1999.
- *Фарли Джим*. Java и распределенные вычисления (Java Distributed Computing). — Себастопол: O'Reilly, 1998.
- Oracle: http://www.oracle.com/technetwork/java/index.html.

Примеры кода, использованного в этой главе

Приведенный в этой главе код был написан на Java и C# .NET.

Глава 15
Шаблоны проектирования

Одна из любопытных особенностей разработки программного обеспечения заключается в том, что при создании программной системы вы фактически моделируете реальную систему. Например, в случае с индустрией информационных технологий можно с уверенностью сказать, что эти технологии *являются* бизнесом — или, по крайней мере, они *претворяют в жизнь* бизнес. Чтобы создавать программные бизнес-системы, разработчики должны хорошо понимать бизнес-модели. Вследствие этого они зачастую гораздо лучше знают бизнес-процессы компаний.

Мы уже сталкивались к этой концепцией на всем протяжении книги, поскольку она связана с нашими образовательными исследованиями. Например, когда мы говорили об использовании наследования для абстрагирования поведений и атрибутов классов млекопитающих, соответствующая модель базировалась на действительной, реальной модели, а не вымышленной, которую мы создали для собственных целей.

Таким образом, создав класс `Mammal`, мы можем применять его для генерирования бесчисленного множества других классов, например `Dog`, `Cat` и т. д., поскольку все классы млекопитающих будут совместно использовать определенные поведения и атрибуты. Такой подход работает, если речь идет о `Dog`, `Cat`, `Squirrel` и других классах млекопитающих, поскольку мы видим шаблоны. Эти шаблоны позволяют нам проинспектировать тот или иной класс, представляющий животных, и решить, действительно ли он является классом млекопитающих или, возможно, классом рептилий, для которых потребуются другие шаблоны, касающиеся поведений и атрибутов.

На всем протяжении истории люди использовали шаблоны во многих областях своей деятельности, включая проектирование. Эти шаблоны идут рука об руку со «Святым Граалем» разработки программного обеспечения, которым является повторное использование программного кода. В этой главе мы рассмотрим шаблоны проектирования — относительно новую область в разработке программного обеспечения (первая книга, посвященная шаблонам проектирования, была издана в 1995 году).

Шаблоны проектирования, пожалуй, являются самыми важными достижениями участников объектно-ориентированного движения за несколько последних лет. Они идеально сочетаются с концепцией разработки программного обеспечения, пригодного для повторного использования. Поскольку объектно-ориентированная разработка всецело связана с повторным использованием, шаблоны прекрасно в нее вписываются.

Базовая концепция шаблонов проектирования «вращается» вокруг принципа наиболее оптимальных методик. Под использованием *наиболее оптимальных методик* мы подразумеваем, что при создании хороших и эффективных решений они документируются таким образом, что другие разработчики смогут извлечь пользу из того, чего кто-то успешно добился ранее.

Одна из важнейших книг, посвященных объектно-ориентированной разработке программного обеспечения, называется «Приемы объектно-ориентированного проектирования. Паттерны проектирования» *(Design Patterns: Elements of Reusable Object-Oriented Software)*, авторами которой выступили Эрих Гамма (Erich Gamma), Ричард Хелм (Richard Helm), Ральф Джонсон (Ralph Johnson) и Джон Влиссидес (John Vlissides). Появление этой книги стало важным событием в индустрии ПО, при этом термины из нее настолько укоренились в лексиконе, используемом в сфере информатики, что авторы приобрели известность как «Банда четырех» (GoF — Gang of Four).

Цель этой главы — объяснить, что такое шаблоны проектирования (подробное описание каждого шаблона проектирования выходит далеко за рамки этой книги и заняло бы не один том). Для этого мы исследуем каждую из трех категорий шаблонов проектирования (порождающие, структурные и поведенческие), как определено «Бандой четырех», и рассмотрим конкретный пример одного шаблона из каждой категории.

Зачем нужны шаблоны проектирования

Нельзя сказать, что концепция шаблонов проектирования появилась от необходимости в программном обеспечении, пригодном для повторного использования. Фактически основополагающий труд по шаблонам проектирования посвящен возведению зданий и городов. Как отмечает Кристофер Александер (Christopher Alexander) в книге «Язык шаблонов: города, здания, строительство» *(A Pattern Language: Towns, Buildings, Construction)*, «каждый шаблон описывает проблему, которая снова и снова возникает в нашей среде, а также ядро решения этой проблемы таким путем, что вы сможете использовать это решение миллион раз, никогда не делая это дважды одинаковым образом».

ЧЕТЫРЕ ЭЛЕМЕНТА ШАБЛОНА

«Банда четырех» описывает следующие четыре важные характеристики шаблона.

- Название шаблона — идентификатор, который можно использовать для описания проблемы проектирования, ее решений и последствий одним или двумя словами. Незамедлительное присвоение названия шаблону расширяет наш словарь проектировщика. Все это позволяет нам проектировать на более высоком уровне абстрагирования. Располагая словарем шаблонов, мы можем говорить о них с коллегами, упоминать их в своей документации и даже дискутировать на эту тему с самими собой. Так легче обдумывать конструкции и разъяснять их другим людям. Подбор удачных названий — один из самых сложных этапов процесса пополнения нашего каталога.
- Проблема — описывает, когда следует применять шаблон. В данном случае объясняется проблема и ее суть. Могут очерчиваться определенные проблемы проектирования, например представление алгоритмов как объектов. Могут характеризоваться классовые или объектные структуры, которые являются признаком негибкой конструкции. Иногда проблема будет включать список условий, которые должны быть предварительно соблюдены, чтобы применение шаблона имело смысл.
- Решение — описывает элементы, образующие конструкцию, их отношения, функции и взаимодействия. Решение не описывает специфическую, конкретную конструкцию или реализацию, поскольку шаблон — это образец, который может применяться во

многих ситуациях. Вместо этого шаблон обеспечивает абстрактное описание проблемы проектирования, а также того, как общее расположение элементов (классов и объектов в нашем случае) позволяет решить ее.

- Последствия — результаты и компромиссные решения в случае применения шаблона. Несмотря на то что последствия часто не озвучиваются, при описании проектных решений они имеют критическое значение для оценки вариантов этих решений, позволяя понять, какие издержки и выгоды повлечет применение шаблона. Когда речь идет о программном обеспечении, то последствия часто связаны с компромиссными решениями в плане места и времени. Они также могут касаться вопросов языка и реализации. Поскольку повторное использование часто свойственно объектно-ориентированному проектированию, последствия применения шаблона включают влияние на гибкость системы, ее расширяемость и переносимость. Перечисление последствий явным образом поможет вам понять и оценить их.

Парадигма «Модель/Вид/Контроллер» в Smalltalk

Ради исторической справки нам нужно рассмотреть парадигму «Модель/Вид/Контроллер» (Model/View/Controller — MVC), представленную в Smalltalk (а также используемую в других объектно-ориентированных языках). Парадигма «Модель/Вид/Контроллер» часто применяется для иллюстрирования истоков шаблонов проектирования. Именно она задействуется при создании интерфейсов пользователя на Smalltalk. Пожалуй, Smalltalk стал первым *популярным* объектно-ориентированным языком.

SMALLTALK

Smalltalk — это результат применения ряда отличных идей от Xerox PARC. Он оказался замечательным языком, который обеспечил фундамент для всех последующих объектно-ориентированных языков. Одна из жалоб на C++ заключается в том, что он, в отличие от Smalltalk, не является подлинным объектно-ориентированным языком. Хотя у C++ было больше последователей в годы зарождения объектно-ориентированного подхода, у Smalltalk всегда была базовая группа очень преданных сторонников.

В книге «Язык шаблонов: города, здания, строительство» *(A Pattern Language: Towns, Buildings, Construction)* компоненты парадигмы «Модель/Вид/Контроллер» определяются следующим образом:

> «Модель является объектом приложения, вид — это экранное представление, а контроллер определяет, как пользовательский интерфейс будет реагировать на вводимые пользователем данные».

Проблема с предшествующими парадигмами заключалась в том, что модель, вид и контроллер смешивались в общую массу, в единое целое. Например, один объект включал бы все три этих компонента. В парадигме «Модель/Вид/Контроллер» эти три компонента обладают раздельными, индивидуальными интерфейсами. Поэтому, если вы захотите изменить пользовательский интерфейс приложения, вам потребуется лишь модифицировать вид. На рис. 15.1 показано, как выглядит парадигма «Модель/Вид/Контроллер».

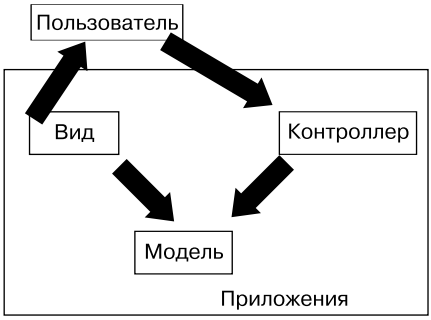

Рис. 15.1. Парадигма «Модель/Вид/Контроллер»

Помните, что в объектно-ориентированной разработке многое имеет отношение к интерфейсам и реализации. Нам необходимо максимально обособить интерфейс от реализации. Нам также нужно максимально обособить интерфейсы друг от друга. Например, мы не хотим смешивать множественные интерфейсы, которые никак не связаны между собой (или с имеющимся решением определенной проблемы). Парадигма «Модель/Вид/Контроллер» стала одной из первых, позволяющих разделять интерфейсы. Она явным образом определяет интерфейсы между специфическими компонентами, которые связаны с широко распространенной и основной проблемой программирования — созданием интерфейсов пользователя и их связыванием с бизнес-логикой и данными, стоящими за ними.

Если вы будете придерживаться парадигмы «Модель/Вид/Контроллер» и станете разделять интерфейс пользователя, бизнес-логику и данные, то ваша система получится гибкой и надежной. Предположим, что интерфейс пользователя находится на клиентском компьютере, бизнес-логика — на сервере приложений, а информация — на сервере данных. Разработка вашего приложения таким путем позволила бы вам изменить внешний вид графического интерфейса пользователя, не затронув бизнес-логику или данные. Аналогичным образом, если вам потребуется изменить свою бизнес-логику и вычислять значение определенного поля по-другому, вы сможете модифицировать ее без необходимости изменять графический интерфейс пользователя. И наконец, если вы захотите заменить базы данных и хранить свою информацию по-другому, то у вас будет возможность изменить подход к хранению информации на сервере данных, не затронув при этом ни графический интерфейс пользователя, ни бизнес-логику. В данном случае, конечно же, предполагается, что интерфейсы между этими тремя компонентами не изменятся.

ПРИМЕР С ИСПОЛЬЗОВАНИЕМ ПАРАДИГМЫ «МОДЕЛЬ/ВИД/КОНТРОЛЛЕР»

В качестве дополнительного примера, скажем поля списка, представьте себе графический интерфейс пользователя, включающий список телефонных номеров. Видом будет поле списка, моделью — список телефонных номеров, а контроллером — логика, связывающая поле списка со списком телефонных номеров.

НЕДОСТАТКИ ПАРАДИГМЫ «МОДЕЛЬ/ВИД/КОНТРОЛЛЕР»

Хотя парадигма «Модель/Вид/Контроллер» представляет собой отличный подход при проектировании, он может оказаться сложным в том смысле, что приходится заблаго-

временно уделять много внимания различным вещам. В целом это проблема, которая имеет место в случае с объектно-ориентированным проектированием, — существует тонкая грань между грамотным подходом к проектированию и таким, который приводит к созданию громоздкой конструкции. Остается вопрос: насколько сложной вам следует сделать систему, полностью спроектировав ее?

Типы шаблонов проектирования

В книге «Язык шаблонов: города, здания, строительство» *(A Pattern Language: Towns, Buildings, Construction)* представлено 23 шаблона, которые разделены на категории, перечисленные ниже. Большинство из этих примеров написано на C++, а некоторые созданы с применением Smalltalk. Время публикации этой книги говорит о том, что тогда уже использовались C++ и Smalltalk. В год издания — 1995-й — интернет-революция как раз достигла своего пика, а язык программирования Java обрел соответствующую популярность. После того как преимущества шаблонов проектирования стали очевидны, авторы поспешили написать множество других книг, стремясь насытить новый рынок. Многие из последующих книг были написаны с ориентиром на использование языка Java.

Так или иначе, неважно, каким фактически окажется используемый язык. Книга «Язык шаблонов: города, здания, строительство» *(A Pattern Language: Towns, Buildings, Construction)*, по сути, посвящена проектированию, а шаблоны могут быть реализованы с применением любого количества языков. Авторы этой книги разделили шаблоны на три категории.

- *Порождающие* — создают за вас объекты, чтобы вам не пришлось заниматься непосредственным созданием экземпляров объектов. Это позволит вашей программе проявлять бо́льшую гибкость при решении о том, какие объекты надлежит создать в конкретном случае.
- *Структурные* — помогают вам составлять из групп объектов более крупные структуры вроде комплексных интерфейсов пользователя или учетных данных.
- *Поведенческие* — помогают вам определять взаимодействие между объектами в вашей системе, а также способ управления потоком в комплектной программе.

Далее мы рассмотрим по одному примеру из каждой названной категории, чтобы вы смогли узнать, что именно представляют собой шаблоны проектирования. Исчерпывающий перечень, а также описание отдельных шаблонов проектирования вы найдете в книгах, список которых приведен в конце этой главы.

Порождающие шаблоны

К числу порождающих шаблонов относятся следующие:

- Abstract factory (Абстрактная фабрика);
- Builder (Строитель);
- Factory method (Фабричный метод);
- Prototype (Прототип);
- Singleton (Одиночка).

Как уже отмечалось ранее, в этой главе мы попробуем разобраться, что такое шаблоны проектирования, а не будем пытаться охарактеризовать все без исключения шаблоны из книги «Банды четырех». Поэтому мы рассмотрим по одному шаблону из каждой категории. Учитывая это, обратимся к примеру порождающего шаблона и взглянем на шаблон Singleton (Одиночка).

Шаблон проектирования Singleton (Одиночка). Этот шаблон (рис. 15.2) является порождающим и используется для регулирования создания объектов, начиная с класса и заканчивая одним объектом. Например, если у вас есть сайт, который включает объект `Counter` для подсчета количества посещений вашего сайта, то вам, конечно же, не понадобится, чтобы при каждом посещении вашей веб-страницы создавался новый экземпляр объекта `Counter`. Вы захотите, чтобы экземпляр этого объекта создавался только при первом посещении, а после этого использовался существующий объект для увеличения значения счетчика.

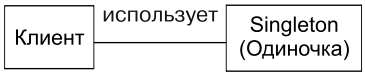

Рис. 15.2. Модель на основе Singleton (Одиночка)

Хотя могут быть и другие способы, позволяющие регулировать создание объектов, зачастую лучше всего позволить классу самому позаботиться о решении этой проблемы. Однако во многих ситуациях использование внешней фабрики оказывается полезным или даже необходимым — в частности, там, где важны такие шаблоны, как Factory (Фабрика), Abstract Factory (Абстрактная фабрика) и Bridge (Мост).

ПРИМЕЧАНИЕ

Помните, что одно из важнейших объектно-ориентированных правил заключается в том, что объект должен отвечать за себя. Это означает, что вопросы, касающиеся жизненного цикла класса, должны решаться в рамках этого класса, а не делегироваться языковым конструкциям вроде static и т. д.

На рис. 15.3 показана UML-модель `Singleton`, взятая непосредственно из книги «Приемы объектно-ориентированного проектирования. Паттерны проектирования» *(Design Patterns: Elements of Reusable Object-Oriented Software)*. Обратите внимание на свойство `uniqueInstance`, которое является статическим объектом `Singleton`, а также на метод `Instance()`. Другие свойства и методы располагаются там, указывая на то, что они потребуются для поддержки бизнес-логики соответствующего класса.

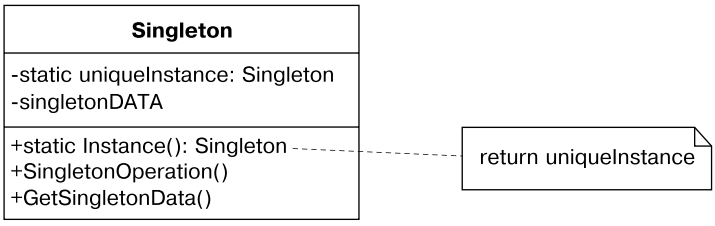

Рис. 15.3. UML-диаграмма Singleton

Любому другому классу, которому потребуется доступ к экземпляру `Singleton`, придется прибегнуть к методу `Instance()`. Создание объекта должно контролироваться конструктором точно так же, как при объектно-ориентированном проектировании чего-либо другого. Мы можем сделать так, что клиенту потребуется прибегнуть к методу `Instance()`, который он затем использует для вызова конструктора.

Приведенный далее Java-код демонстрирует общий вариант `Singleton`:

```
public class ClassicSingleton {
  private static ClassicSingleton instance = null;

  protected ClassicSingleton() {
    // Существует только для того, чтобы помешать созданию экземпляров
  }
  public static ClassicSingleton getInstance() {
    if(instance == null) {
      instance = new ClassicSingleton();
    }
    return instance;
  }
}
```

Мы можем создать более конкретный пример с образчиком `Counter` для веб-страницы, который был использован нами ранее:

```
public class Counter
{
  private int counter;
  private static Counter instance = null;

  protected Counter()
  {
  }

  public static Counter getInstance() {
    if(instance == null) {
      instance = new Counter ();
      System.out.println("Новый экземпляр создан\n");
    }
    return instance;
  }

  public void incrementCounter()
  {
    counter++;
  }

  public int getCounter()
  {
    return(counter);
  }

}
```

Главное, что следует отметить в этом коде, — он регулирует создание объектов. Может быть создан только один объект Counter. Код для этого выглядит так:

```
public static Counter getInstance() {
  if(instance == null) {
    instance = new Counter ();
    System.out.println("Новый экземпляр создан\n");
  }
  return instance;
}
```

Обратите внимание, что, если instance присвоено значение null, это подразумевает, что экземпляр объекта еще предстоит создать. В данной ситуации создается новый объект Counter. Если значением instance не является null, то это говорит о том, что был создан экземпляр объекта Counter, а новые объекты создавать не потребуется. В данном случае ссылка на единственный доступный объект возвращается приложению.

БОЛЕЕ ОДНОЙ ССЫЛКИ

Вполне допускается несколько ссылок на Singleton. Если вы создадите ссылки в приложении, каждая из которых будет вести к Singleton, то вам придется управлять множественными ссылками.

Хотя этот код, несомненно, интересен, важно понимать, как происходит создание экземпляра Singleton и каким образом им управляет приложение. Взгляните на такой код:

```
public class Singleton
{
  public static void main(String[] args)
  {
    Counter counter1 = Counter.getInstance();
    System.out.println("Счетчик : " + counter1.getCounter() );

    Counter counter2 = Counter.getInstance();
    System.out.println("Счетчик : " + counter2.getCounter() );

  }
}
```

ДВЕ ССЫЛКИ НА ОДИН COUNTER

Знайте, что в этом примере к Counter ведут две отдельные ссылки. Таким образом, когда значение Counter изменится, обе эти ссылки будут отражать соответствующее обновление.

Этот код задействует Singleton с Counter. Посмотрите, как создаются объекты:

```
Counter counter1 = Counter.getInstance();
```

Здесь конструктор не используется. Создание экземпляра объекта контролируется методом getInstance(). На рис. 15.4 показано, что произойдет при выполнении этого кода. Обратите внимание, что сообщение Новый экземпляр создан выводится

только один раз. При создании counter2 он получает копию оригинального объекта — так же, как counter1.

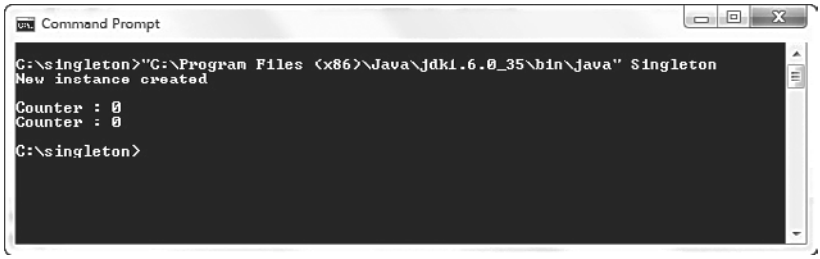

Рис. 15.4. Использование Singleton с Counter

Убедимся в том, что ссылки для counter1 и counter2 одинаковы. Мы можем обновить программный код следующим образом:

```
public class Singleton
{
  public static void main(String[] args)
  {
    Counter counter1 = Counter.getInstance();
    counter1.incrementCounter();
    counter1.incrementCounter();
    System.out.println("Счетчик : " + counter1.getCounter() );

    Counter counter2 = Counter.getInstance();
    counter2.incrementCounter();
    System.out.println("Счетчик : " + counter2.getCounter() );

  }
}
```

На рис. 15.5 показан вывод приложения Singleton. Обратите внимание, что в данной ситуации мы увеличиваем значение counter1 вдвое, поэтому значение счетчика будет равно 2. После создания ссылки для counter2 она ведет к тому же объекту, что и ссылка для counter1, поэтому при увеличении значения счетчика оно теперь будет равно 3 (2 + 1).

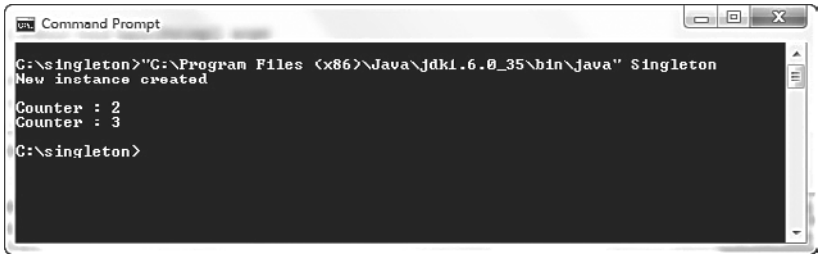

Рис. 15.5. Использование обновленного Singleton с Counter

ЕЩЕ РАЗ О ПАКЕТНЫХ ФАЙЛАХ

Как уже демонстрировалось в предыдущих главах, для облегчения тестирования простых, но функциональных образовательных примеров такого рода мне нравится создавать старомодные пакетные файлы, чтобы сделать ввод путей к Java-классам более точным. Вы можете запустить приложение Singleton, поместив приведенный далее код в пакетный файл Singleton.bat, а затем введя Singleton в командной строке:

```
"C:\Program Files (x86)\Java\jdk1.6.0_35\bin\java" Singleton
```

Структурные шаблоны

Структурные шаблоны используются для создания более крупных конструкций из групп объектов. Приведенные далее семь шаблонов проектирования относятся к категории структурных:

- Adapter (Адаптер);
- Bridge (Мост);
- Composite (Компоновщик);
- Decorator (Декоратор);
- Facade (Фасад);
- Flyweight (Приспособленец);
- Proxy (Заместитель).

В качестве примера из категории структурных шаблонов рассмотрим Adapter (Адаптер). Это также один из важнейших шаблонов проектирования. Он представляет собой хороший пример того, как разделяются реализация и интерфейс.

Шаблон проектирования Adapter (Адаптер). Данный шаблон — это инструмент, с помощью которого вы можете создать отличающийся интерфейс для уже существующего класса. Этот шаблон по сути предусматривает создание классовой обертки. Другими словами, вы создадите новый класс, который объединит (обернет собой) функциональность существующего класса с новым и — в идеале — лучшим интерфейсом. Простой пример обертки — Java-класс `Integer`, который обертывает собой одно целочисленное значение. Вы можете спросить, зачем так делать. Помните, что в объектно-ориентированной системе все сущности являются объектами. В Java примитивы, например целые, дробные числа и т. д., не являются объектами. Если вам понадобится выполнить над ними какое-либо действие, скажем преобразование, то придется обращаться с ними как с объектами. Вы создадите объект-обертку и обернете им соответствующий примитив. Таким образом, вы сможете взять примитив вроде следующего:

```
int myInt = 10;
```

и обернуть его объектом `Integer`:

```
Integer myIntWrapper = new Integer (myInt);
```

Теперь можно выполнить преобразование, чтобы мы смогли обращаться с ним как со строкой:

```
String myString = myIntWrapper.toString();
```

Эта обертка дает нам возможность обращаться с исходным целочисленным значением как с объектом, благодаря чему обеспечиваются все преимущества объектов.

В том, что касается шаблона Adapter (Адаптер) как такового, рассмотрим пример интерфейса `MailTool`. Предположим, что вы приобрели код, который обеспечивает всю функциональность, требуемую вам для реализации почтового клиента. Этот инструмент предоставляет все, что вам нужно от почтового клиента, с тем исключением, что вы хотели бы немного изменить интерфейс. Фактически все, что вы хотите сделать, — это изменить API-интерфейс для извлечения своей электронной почты.

Приведенный далее класс — очень простой образец почтового клиента для этого примера:

```
package MailTool;
public class MailTool {
  public MailTool () {
  }
  public int retrieveMail() {

    System.out.println ("Вам пришла почта");

    return 0;
  }
}
```

При вызове метода `retrieveMail()` ваша почта откликается с использованием очень оригинального приветствия `Вам пришла почта`. Теперь предположим, что вы захотели изменить интерфейс всех клиентов в вашей компании с `retrieveMail()` на `getMail()`. Вы можете создать интерфейс, чтобы претворить это в жизнь:

```
package MailTool;
interface MailInterface {
   int getMail();
}
```

Теперь вы можете создать свой `MailTool`, который будет оберткой для оригинального `MailTool`, и предусмотреть собственный интерфейс:

```
package MailTool;
class MyMailTool implements MailInterface {
  private MailTool yourMailTool;
  public MyMailTool () {
    yourMailTool= new MailTool();
     setYourMailTool(yourMailTool);
  }
  public int getMail() {
    return getYourMailTool().retrieveMail();
  }
  public MailTool getYourMailTool() {
    return yourMailTool ;
  }
  public void setYourMailTool(MailTool newYourMailTool) {
```

```
      yourMailTool = newYourMailTool;
    }
  }
```

Внутри этого класса вы создаете экземпляр оригинального `MailTool`, который хотите модифицировать. Данный класс реализует `MailInterface`, из-за чего вам приходится реализовать метод `getMail()`. Внутри этого метода вы буквально вызываете метод `retrieveMail()` оригинального `MailTool`.

Чтобы использовать свой новый класс, вам потребуется создать экземпляр собственного нового `MailTool` и вызвать метод `getMail()`:

```
package MailTool;
public class Adapter
{
  public static void main(String[] args)
  {
    MyMailTool myMailTool = new MyMailTool();

    myMailTool.getMail();

  }
}
```

При вызове метода `getMail()` вы будете использовать этот новый интерфейс для вызова метода `retrieveMail()` оригинального `MailTool`. Это очень простой пример. Вместе с тем, создав данную обертку, вы расширите интерфейс и добавите собственную функциональность в оригинальный класс.

Концепция шаблона Adapter (Адаптер) довольно проста, однако вы сможете создавать новые и мощные интерфейсы, используя этот шаблон.

Поведенческие шаблоны

К числу поведенческих шаблонов относятся следующие:

- Chain of response (Цепочка ответственности);
- Command (Команда);
- Interpreter (Интерпретатор);
- Iterator (Итератор);
- Mediator (Посредник);
- Memento (Хранитель);
- Observer (Наблюдатель);
- State (Состояние);
- Strategy (Стратегия);
- Template method (Шаблонный метод);
- Visitor (Посетитель).

В качестве примера из категории поведенческих шаблонов взглянем на шаблон Iterator (Итератор). Это один из наиболее широко используемых шаблонов, который в то же время реализован несколькими языками программирования.

Шаблон проектирования Iterator (Итератор). Итераторы представляют собой стандартный механизм обхода коллекций, например векторов. При этом должна обеспечиваться функциональность, благодаря которой будет возможен доступ к каждому элементу коллекции поодиночке. Шаблон Iterator (Итератор) обеспечивает скрытие информации, сохраняя защищенной внутреннюю структуру коллекции. Этот шаблон также предполагает, что можно создать несколько итераторов и они не станут мешать друг другу. Java предусматривает собственную реализацию шаблона Iterator (Итератор). Приведенный далее код создает вектор, после чего вставляет в него несколько строк:

```
package Iterator;
import java.util.*;
public class Iterator {
  public static void main(String args[]) {

    // Создать экземпляр ArrayList
    ArrayList<String> names = new ArrayList();

    // Добавить значения в ArrayList
    names.add(new String("Джо"));
    names.add(new String("Мэри"));
    names.add(new String("Боб"));
    names.add(new String("Сью"));

    // Теперь осуществить итерацию по именам
    System.out.println("Имена:");
    iterate(names );
  }

  private static void iterate(ArrayList<String> arl) {
    for(String listItem : arl) {
      System.out.println(listItem.toString());
    }
  }
}
```

Затем мы создаем перечисление, чтобы можно было осуществить по нему итерацию. Для обеспечения функциональности, связанной с выполнением итераций, предусмотрен метод `iterate()`. В нем мы используем Java-метод перечисления `hasMoreElements()`, который позволяет выполнить обход вектора и показать все имена.

Антишаблоны

В то время как шаблоны проектирования развиваются из успешных практик, *антишаблоны* можно представлять себе как наборы неудавшихся методик. Существуют убедительные документальные доказательства того, что большинство программных проектов в конечном счете признаются неудачными. Фактически, как отмечается в статье «Создавая хаос» *(Creating Chaos)*, написанной Джонни Джонсоном (Johnny

Johnson), по меньшей мере одна треть всех проектов полностью отменяется. По-видимому, многие из этих неудач обусловлены плохими проектными решениями.

Термин «антишаблон» стал следствием того факта, что шаблоны проектирования создаются с целью заранее решать проблемы определенного типа. Антишаблон, с другой стороны, является реакцией на проблему и происходит из плохого опыта. Одним словом, в то время как шаблоны проектирования базируются на надежных методиках проектирования, антишаблоны можно рассматривать как методики, которых следует избегать.

В выпуске журнала *C++ Report* за ноябрь 1995 года Эндрю Кениг (Andrew Koenig) охарактеризовал два варианта антишаблонов:

- описывающие плохое решение проблемы, которое приводит к возникновению скверной ситуации;
- объясняющие, как выйти из скверной ситуации, а затем выработать хорошее решение.

Многие люди считают, что антишаблоны полезнее шаблонов проектирования, так как призваны решать проблемы, которые уже возникли. В данном случае все сводится к концепции анализа первопричин. Можно провести исследование с использованием данных, которое способно показать, почему первоначальная конструкция или, возможно, текущий шаблон проектирования не позволили добиться требуемой цели. Таким образом, антишаблоны обладают преимуществом в виде ретроспективного взгляда.

Например, в своей статье «Повторное использование шаблонов и антишаблонов» *(Reuse Patterns and Antipatterns)* Скотт Амблер приводит шаблон Robust Artifact (Надежный артефакт) и определяет его следующим образом:

> «Инструмент, который хорошо документирован, создан отвечать общим требованиям, а не специфичным для определенного проекта, как следует протестирован и предусматривает несколько примеров того, как с ним работать. Намного более вероятно, что повторно будут использоваться инструменты, обладающие такими качествами, а не те, которые их лишены. Robust Artifact (Надежный артефакт) — это инструмент, легкий для понимания и простой в эксплуатации».

Однако, несомненно, бывает много ситуаций, когда то или иное решение объявляется пригодным для повторного использования, однако затем никто так никогда и не прибегает к нему снова. Поэтому, чтобы пояснить антишаблон, Скотт Амблер пишет:

> «Сторонний разработчик должен проанализировать Reuseless Artifact (Артефакт, никем не используемый повторно), чтобы выяснить, заинтересует ли он кого-нибудь. Если да, то его надлежит переработать так, чтобы он превратился в Robust Artifact (Надежный артефакт)».

Таким образом, антишаблоны приводят к пересмотру существующих конструкций и их непрерывному рефакторингу до тех пор, пока не будет найдено приемлемое решение.

Резюме

В той главе мы исследовали концепцию шаблонов проектирования. Шаблоны являются частью повседневной жизни, и именно таким образом вам следует мыслить при объектно-ориентированном проектировании.

В этой главе шаблоны проектирования рассмотрены лишь кратко, а вам следует исследовать эту тему, воспользовавшись одной из книг, приведенных в конце текущей главы.

Ссылки

- *Александер Кристофер и др.* Язык шаблонов: города, здания, строительство (A Pattern Language: Towns, Buildings, Construction). — Оксфорд: Oxford University Press, 1977.
- *Гамма Эрих и др.* Приемы объектно-ориентированного проектирования. Паттерны проектирования (Design Patterns: Elements of Reusable Object-Oriented Software). — Бостон: Addison-Wesley, 1995.
- *Ларман Крейг.* Применение UML и шаблонов проектирования. Введение в объектно-ориентированный анализ, проектирование и итеративную разработку (Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development). 3-е изд. — Хобокен: Wiley, 2004.
- *Гранд Марк.* Шаблоны проектирования в Java. Каталог популярных шаблонов проектирования, проиллюстрированных с помощью UML (Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML). 2-е изд., — Хобокен: Wiley, 2002. — Т. 1.
- *Амблер Скотт.* Повторное использование шаблонов и антишаблонов (Reuse Patterns and Antipatterns) / Software Development Magazine. — 2000.
- *Яворски Джейми.* Платформа Java 2 в действии (Java 2 Platform Unleashed). — Индианаполис: Sams Publishing, 1999.
- *Джонсон Джонни.* Создавая хаос (Creating Chaos) / American Programmer. — Июль 1995.

Примеры кода, использованного в этой главе

Приведенный далее код написан на C# .NET. Эти примеры соответствуют Java-коду, продемонстрированному в текущей главе.

C# .NET

```
Counter.cs
using System;
using System.Collections.Generic;
using System.Text;
```

```
namespace Counter
{
  class Counter
  {
    private int counter;
    private static Counter instance = null;

    protected Counter()
    {

    }

    public static Counter getInstance()
    {
      if (instance == null)
      {
        instance = new Counter();
        Console.WriteLine("Новый экземпляр Counter...");
      }
      return instance;
    }

    public void incrementCounter()
    {
      counter++;
    }

    public int getCounter()
    {
      return counter;
    }
  }
}
```

Singleton.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace Counter
{
  class Singleton
  {
    public Singleton()
    {
      Counter counter1 = Counter.getInstance();
      counter1.incrementCounter();
      counter1.incrementCounter();
      Console.WriteLine("Счетчик = " + counter1.getCounter());
```

```
      Counter counter2 = Counter.getInstance();
      counter2.incrementCounter();
      Console.WriteLine("Счетчик = " + counter2.getCounter());
      Console.WriteLine("Нажмите любую клавишу, чтобы продолжить...");
      Console.ReadKey();
    }
  }
}
```

MailTool.cs

```
using System;

namespace MailAdapter
{

  class MailTool
  {
    public MailTool()
    {

    }

    public int retrieveMail()
    {
      Console.WriteLine("Вам пришла почта!");
      return 0;
    }
  }
}
```

Mailinterface.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace MailAdapter
{
  interface MailInterface
  {
    int getMail();
  }
}
```

MyMail.cs

```
namespace MailAdapter
{
  class MyMailTool : MailInterface
  {
    private MailTool yourMailTool;
```

```
    public MyMailTool()
    {
      yourMailTool = new MailTool();
      setYourMailTool(yourMailTool);
    }

    public int getMail()
    {
      return getYourMailTool().retrieveMail();
    }

    public MailTool getYourMailTool()
    {
      return yourMailTool;
     }

     public void setYourMailTool(MailTool newYourMailTool)
     {
       yourMailTool = newYourMailTool;
     }
  }
}
```

Adapter.cs

```
using System;
using System.Collections.Generic;
using System.Text;

namespace MailAdapter
{
  class Adapter
  {
    public Adapter()
    {
      MyMailTool myMailTool = new MyMailTool();
      myMailTool.getMail();
      Console.WriteLine();
      Console.WriteLine("Нажмите любую клавишу, чтобы продолжить...");
      Console.ReadKey();
    }
  }
}
```

Iterator.cs

```
using System;
using System.Collections.Generic;
using System.Text;
using System.Collections;
```

```
namespace Iterator
{
  class Iterator
  {
    public Iterator()
    {
      // Создать экземпляр Arraylist
      ArrayList myList = new ArrayList();

      // Добавить значения в список
      myList.Add("Джо");
      myList.Add("Мэри");
      myList.Add("Боб");
      myList.Add("Сью");

      // Осуществить итерацию по элементам
      Console.WriteLine("Имена:");
      iterate(myList);
    }

    static void iterate(ArrayList arl)
    {
      foreach (String listItem in arl)
      {
        Console.WriteLine(listItem);
      }
    }
  }

}
```

М. Вайсфельд

Объектно-ориентированное мышление

Перевел с английского В. Черник

Заведующий редакцией	*Д. Виницкий*
Ведущий редактор	*Н. Гринчик*
Художник	*Л. Адуевская*
Корректоры	*Т. Курьянович, Е. Павлович*
Верстка	*А. Барцевич*

ООО «Питер Пресс», 192102, Санкт-Петербург, ул. Андреевская (д. Волкова), д. 3, литер А, пом. 7Н.

Налоговая льгота — общероссийский классификатор продукции ОК 005-93, том 2; 95 3005 — литература учебная.

Подписано в печать 28.02.14. Формат 70×100/16. Усл. п. л. 24,510. Тираж 1500. Заказ 0000.

Отпечатано в полном соответствии с качеством предоставленных издательством материалов
в ГППО «Псковская областная типография». 180004, Псков, ул. Ротная, 34.