Мэтт Вайсфельд

5-е международное издание

Объектно-ориентированный подход

The Object-Oriented Thought Process

Fifth Edition

Matt Weisfeld

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi
Mexico City • São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Объектно-ориентированный подход

5-е международное издание

Мэтт Вайсфельд

ПИТЕР®

Санкт-Петербург • Москва • Екатеринбург • Воронеж
Нижний Новгород • Ростов-на-Дону
Самара • Минск

2020

ББК 32.973.2-018
УДК 004.42
В14

Вайсфельд Мэтт

В14 Объектно-ориентированный подход. 5-е межд. изд. — СПб.: Питер, 2020. — 256 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1431-3

Объектно-ориентированное программирование (ООП) лежит в основе языков C++, Java, C#, Visual Basic .NET, Ruby, Objective-C и даже Swift. Не могут обойтись без объектов веб-технологии, ведь они используют JavaScript, Python и PHP.

Именно поэтому Мэтт Вайсфельд советует выработать объектно-ориентированное мышление и только потом приступать к объектно-ориентированной разработке на конкретном языке программирования.

Эта книга написана разработчиком для разработчиков и позволяет выбрать оптимальные подходы для решения конкретных задач. Вы узнаете, как правильно применять наследование и композицию, поймете разницу между агрегацией и ассоциацией и перестанете путать интерфейс и реализацию.

Технологии программирования непрерывно меняются и развиваются, но объектно-ориентированные концепции не зависят от платформы и остаются неизменно эффективными. В этом издании основное внимание уделяется фундаментальным основам ООП: паттернам проектирования, зависимостям и принципам SOLID, которые сделают ваш код понятным, гибким и хорошо сопровождаемым.

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

ББК 32.973.2-018
УДК 004.42

ISBN 978-0135181966 англ.
ISBN 978-5-4461-1431-3

СОДЕРЖАНИЕ

ОГЛАВЛЕНИЕ

Шэрон, Стейси, Стефани и Пауло

БЛАГОДАРНОСТИ

Как и в случае с первыми четырьмя изданиями, эта книга потребовала совместных усилий многих людей. Я хотел бы поблагодарить как можно больше этих людей, поскольку без их помощи книга никогда бы не увидела свет.

Прежде всего, хотелось бы выразить благодарность моей жене Шэрон за помощь. Она не только поддерживала и подбадривала меня во время длительного написания книги, но и выступила в роли первого редактора черновиков.

Я также хотел бы поблагодарить маму и других членов моей семьи за их постоянную поддержку.

Трудно поверить в то, что работа над первым изданием этой книги началась еще в 1998 году. На протяжении работы над всеми пятью изданиями мне было очень приятно сотрудничать с ребятами из компании Pearson. А работа с такими редакторами, как Марк Тэбер (Mark Taber) и Тоня Симпсон (Tonya Simpson), принесла мне массу удовольствия.

Выражаю особую благодарность Джону Апчерчу (Jon Upchurch) за проверку большей части приведенного в этой книге кода, а также за научное редактирование рукописи. Его обширные познания в технических областях оказали неоценимую помощь.

И наконец, спасибо моим дочерям Стейси и Стефани, а также коту Пауло за то, что постоянно держат меня в тонусе.

ОБ АВТОРЕ

Мэтт Вайсфельд — профессор колледжа, разработчик программного обеспечения и автор. Проживает в Кливленде, штат Огайо. Прежде чем стать штатным преподавателем, Мэтт 20 лет проработал в индустрии информационных технологий разработчиком ПО, занимая должность адъюнкт-профессора, вел предпринимательскую деятельность. Имеет степень магистра MBA и computer science. Помимо первых четырех изданий книги «Объектно-ориентированный подход», написал еще две книги 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 международном журнале *Project Management*.

Авторские права на иллюстрации

Рисунок 8.1, скриншот Microsoft Word, © Корпорация Microsoft, 2019.

Рисунок 8.2, скриншот документации API, © Корпорация Oracle, 1993, 2018.

От издательства

Ваши замечания, предложения и вопросы отправляйте по адресу электронной почты comp@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 подходу. Хотя выбор темы и названия книги является важным решением, оно оказывается совсем не простым, когда речь идет о концептуальной теме. 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 студенты 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 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.

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 статью, написанную Крейгом Ларманом (Craig

Larman) *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-ориентрованных концепциях, аналогично попытке научиться читать электрические схемы, изначально ничего не зная об электричестве.

Та же проблема возникает с языками программирования. Как отмечалось, многие программисты на C ушли в область объектной ориентации, перейдя на C ++, прежде чем соприкоснуться с объектно-ориентированными концепциями. Это всегда обнаруживается в процессе собеседования. Довольно часто разработчики, которые утверждают, что они программисты на C ++, являются просто программистами на C, использующими компиляторы C ++. Даже сейчас, с такими хорошо зарекомендовавшими себя языками, как C # .NET, VB .NET, Objective-C, Swift и Java, ключевые вопросы 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++. Язык 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, сначала уделите время освоению объектно-ориентированного мышления.

Что нового в пятом издании

Как часто отмечалось во введении к предыдущим изданиям, мое видение первого издания заключалось в том, чтобы придерживаться концепций, а не сосредоточиваться на конкретной новейшей технологии. Несмотря на следование такому же подходу в пятом издании, я стараюсь привести больше «контраргументов», чем в предыдущих четырех изданиях. Этим я даю понять, что хотя объектно-ориентированная разработка, несомненно, имеет важнейшее значение, не стоит ограничивать лишь ею свое внимание.

Поскольку книга была впервые издана в 1999 году, за прошедшее время успели появиться многие технологии, а некоторые были вытеснены. В те времена язык Java только находился в начале своего становления и был главным языком в сфере объектно-ориентированного программирования. В скором времени веб-страницы уже являлись неотъемлемой частью личной и деловой жизни. И нам хорошо известно, насколько повсеместными стали мобильные устройства. За последние 20 лет разработчики программ столкнулись с XML, JSON, CSS, XSLT, SOAP и RESTful Web Services. В устройствах Android используется Java и, уже, Kotlin, а в iOS — Objective C и Swift.

Хочу обратить внимание на то, что за 20 лет (и четыре издания книги) мы охватили много различных технологий. В этом издании я ставлю себе основной задачей сконцентрировать весь предыдущий материал и вернуться к изначальному замыслу первого издания — концепциям объектно-ориентированного программирования. Мне приятно считать, что каким бы ни был успех первого издания книги, он связан с тем, что оно было сосредоточено на самих концепциях. Мы, в некотором роде, прошлись по кругу, потому что настоящее издание включает в себя все технологии, которые были упомянуты.

Наконец, концепции, которые объединяют все эти технологии в методику проектирования, описываются принципами SOLID. Ими пронизана каждая глава этого издания, а также две новые главы в конце.

Пять принципов SOLID:

- **SRP** — принцип единственной ответственности (Single Responsibility Principle).
- **OCP** — принцип открытости/закрытости (Open/Close Principle).
- **LSP** — принцип подстановки Барбары Лисков (Liskov Substitution Principle).
- **ISP** — принцип разделения интерфейса (Interface Segregation Principle).
- **DIP** — принцип инверсии зависимостей (Dependency Inversion Principle).

В первых девяти главах я расскажу о принципах объектно-ориентированного программирования, которые считаю классическими. Последние три главы посвящены паттернам проектирования, избеганию зависимостей и понятию

SOLID, которое построено на классических принципах и предполагает четко определенную методику.

Целевая аудитория

Эта книга представляет собой общее введение в фундаментальные концепции объектно-ориентированного программирования. Понятие *концепции* важно, потому что хотя весь материал темы для наглядности сопровождается кодом, главная цель книги — познакомить читателя с азами объектно-ориентированного мышления. Немаловажно для программистов и понимание того, что объектно-ориентированное программирование не представляет собой какой-то отчетливой парадигмы (пускай многие в это и верят), но является лишь частью обширного инструментария для современных разработчиков программного обеспечения.

Когда в 1995-м был подготовлен начальный материал для первого издания, объектно-ориентированное программирование только появилось. Я могу так сказать потому, что, помимо немногих объектно-ориентированных языков вроде Smalltalk, настоящих языков объектно-ориентированного программирования в то время не существовало. C++, где применение объектно-ориентированных конструкций необязательно, был доминирующим языком программирования, основанным на C. Первый релиз Java 1.0 состоялся в 1996 году, а C# — в 2002-м. Собственно, когда вышло первое издание этой книги в 1999 году, никто не мог быть уверен, что объектно-ориентированное программирование действительно станет ведущей парадигмой среди разработчиков (Java 2 вышел только в декабре 1998-года). Несмотря на доминирование в настоящее время, объектно-ориентированное программирование не лишено некоторых любопытных изъянов, к которым мы еще вернемся.

Итак, аудитория этой книги сейчас отличается от аудитории первого издания.

С 1995 и вплоть до 2010 года я в основном обучал структурных программистов искусству объектно-ориентированного программирования. Подавляющее большинство моих учеников набирали опыт на COBOL, FORTRAN, C и VB как во время учебы, так и на работе. Сегодняшние ученики, которые выпускаются из заведений, пишут видеоигры, создают сайты или мобильные приложения, практически уже научились писать программы на объектно-ориентированных языках. Из этого следует, что подход пятого издания значительно отличается от того, что был в первом, втором и последующих. Вместо того чтобы обучать структурных программистов разработке на объектно-ориентированных языках, сейчас нам нужно обучать поколение программистов, выросших на таких языках.

Целевая аудитория этой книги включает проектировщиков, разработчиков, программистов и менеджеров проектов, дизайнеров, проще говоря, всех желающих получить общее представление о том, что такое объектно-ориентированное программирование. Я очень надеюсь на то, что прочтение этой книги создаст прочную основу для перехода к материалам, затрагивающим более продвинутые темы.

Подход, примененный 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 необязательно 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 и диаграмм класса?

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# и т. д., 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 строго утверждать, что эта книга *не* учит языку Java, C# .NET, VB .NET, Objective-C, Swift 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.

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 использования сети интернет в качестве платформы для бизнеса и развлечений. А после того как стало очевидным, что 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 25 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* (справедливо для того, что я называю классическим объектно-ориентированным программированием). 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 подробно рассмотрим все эти концепции в книге.

Одна из трудностей, с которыми мне пришлось столкнуться 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 следующих главах.) Даже сейчас многие разработчики стараются избегать наследования, насколько это представляется возможным. Вот и встает вопрос: «А стоит ли вообще применять наследование?».

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. Важно помнить, что с наследованием придется иметь дело с огромной вероятностью при сопровождении кода, поэтому изучить его нужно в любом случае.

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 относительно 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, одной из проблем, с которыми сталкивались разработчики, становилась интеграция 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 не стоит этого делать лишь ради 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 25 лет наблюдалось постоянное и значительное увеличение количества объектно-ориентированных разработок, зависимость мирового сообщества от сетей вроде интернета и мобильных инфраструктур способствовала 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 «Проектирование с использованием объектов».

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 востребованы.

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. Ориентируясь на мобильные веб-приложения, 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 способствовало 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 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 в категориях «объекты».

ПОДСКАЗКА

Посмотрите на YouTube видеолекцию гуру объектно-ориентированного программирования Роберта Мартина. На его взгляд, утверждение «люди мыслят объектно» впервые сделали маркетологи. Немного пищи для размышлений.

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 поведение. 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.

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 поведение обычно разделяются.

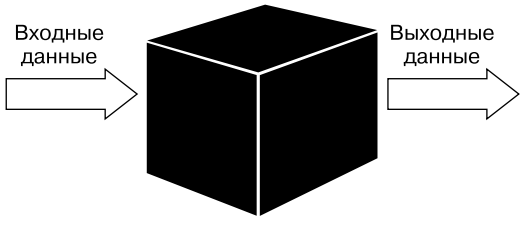

Рис. 1.1. Черный ящик

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.

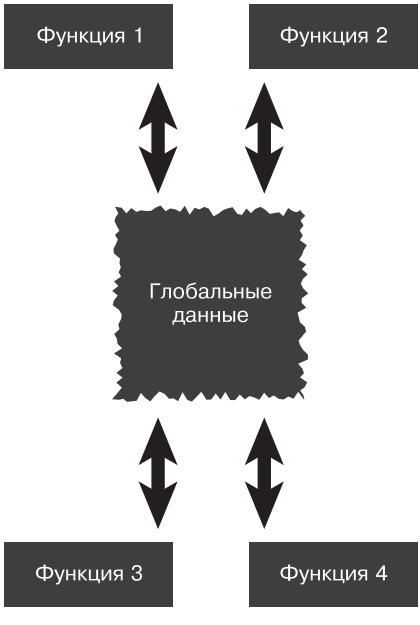

Рис. 1.2. Использование глобальных данных

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 програм-

мы 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.

СОКРЫТИЕ ДАННЫХ

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

Объединив атрибуты и методы в одной сущности (это действие в объектно-ориентированной терминологии 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

Имейте в виду, что можно создать неудачно спроектированные объектно-ориентированные классы, которые не ограничивают доступ к атрибутам классов. Суть заключается 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 объект `myObject` (при условии, что средства для извлечения значения суммы останутся прежними). Все, что вам нужно, — это сумма, и вам *безразлично*, как она вычисляется.

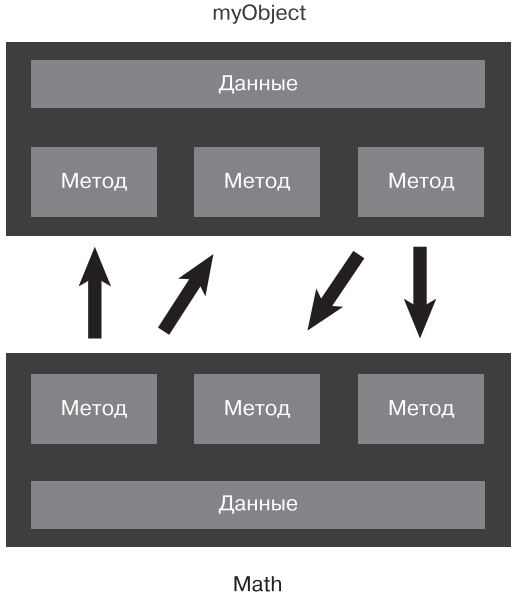

Рис. 1.3. Коммуникации между объектами

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 — в электронном или алгоритмическом порядке.

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, 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.

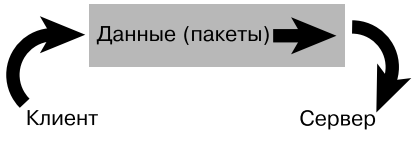

Рис. 1.4. Данные, передаваемые по сети

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.

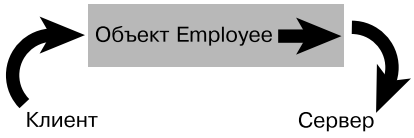

Рис. 1.5. Объект, передаваемый по сети

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.

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

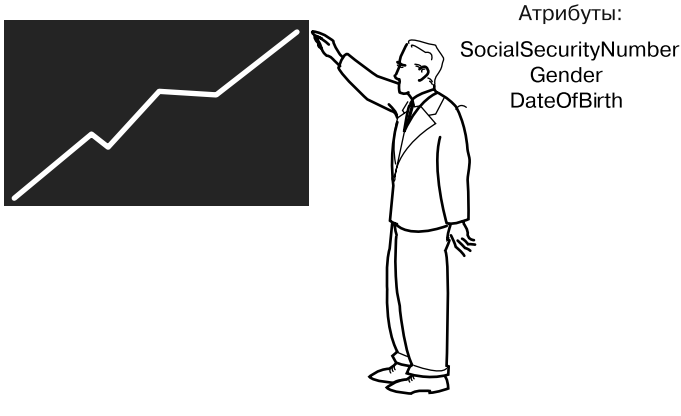

Рис. 1.6. Атрибуты объекта Employee

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.

Поведение 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 содержатся 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. 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 такой технике 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 в последующих главах.

ГЕТТЕРЫ И СЕТТЕРЫ

Концепция геттеров и сеттеров поддерживает концепцию сокрытия 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.

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.

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 «Прочие объектно-ориентированные концепции».

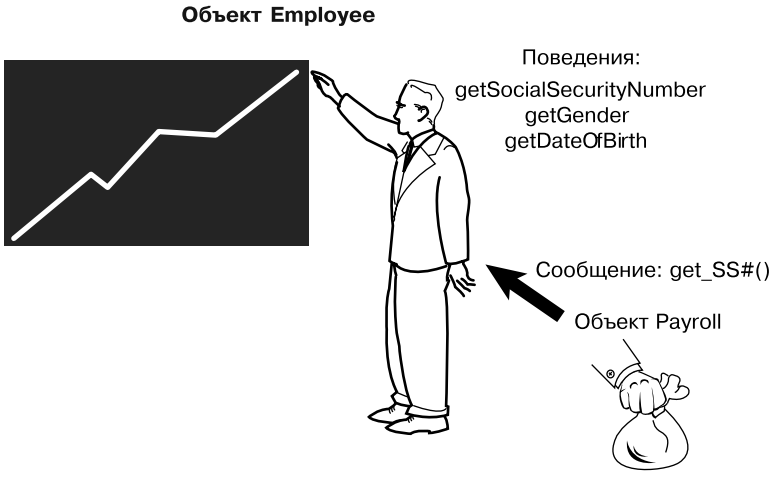

Рис. 1.7. Поведения объекта Employee

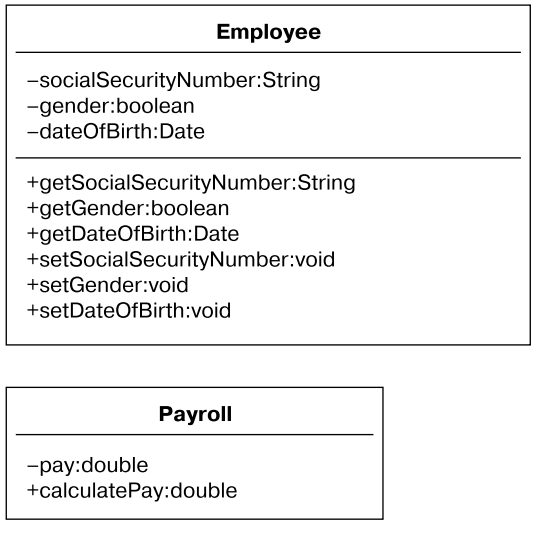

Рис. 1.8. Диаграммы классов Employee и Payroll

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

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 рассматриваются по ходу всей книги. Они 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

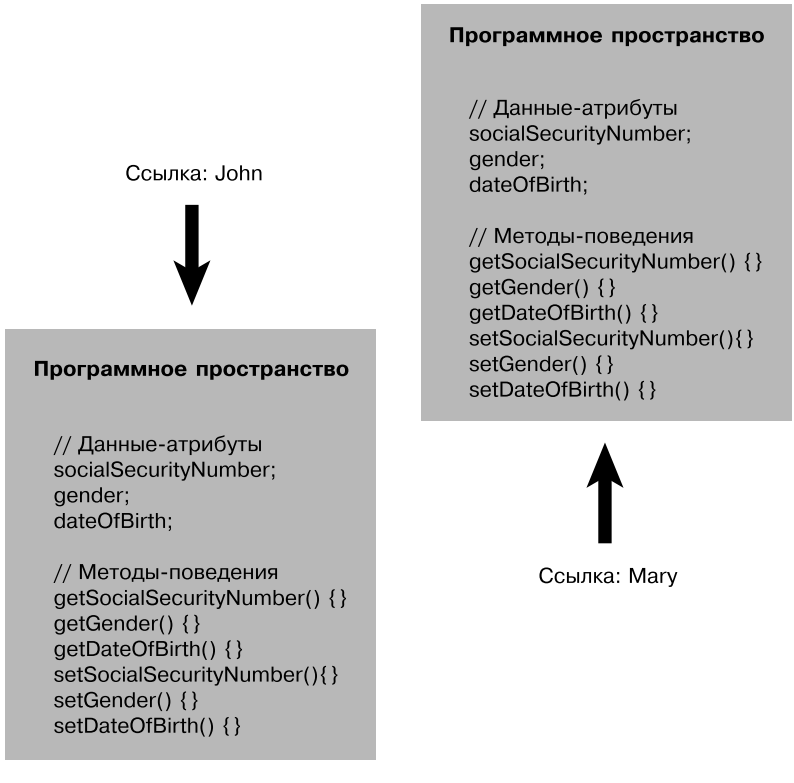

Рис. 1.9. Программные пространства

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`.

ВОПРОС РЕАЛИЗАЦИИ

Знайте, что необязательно 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, какой-либо определенный велосипед — 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.

Несмотря на то что эта книга сосредоточена на концепциях объектно-ориентированного программного обеспечения, а не на конкретной реализации, 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 для некоторых ключевых примеров код предоставляется для загрузки на нескольких языках программирования.

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 кондитерские формочки для объектов, как показано на рис. 1.10. Класс используется для создания объекта.

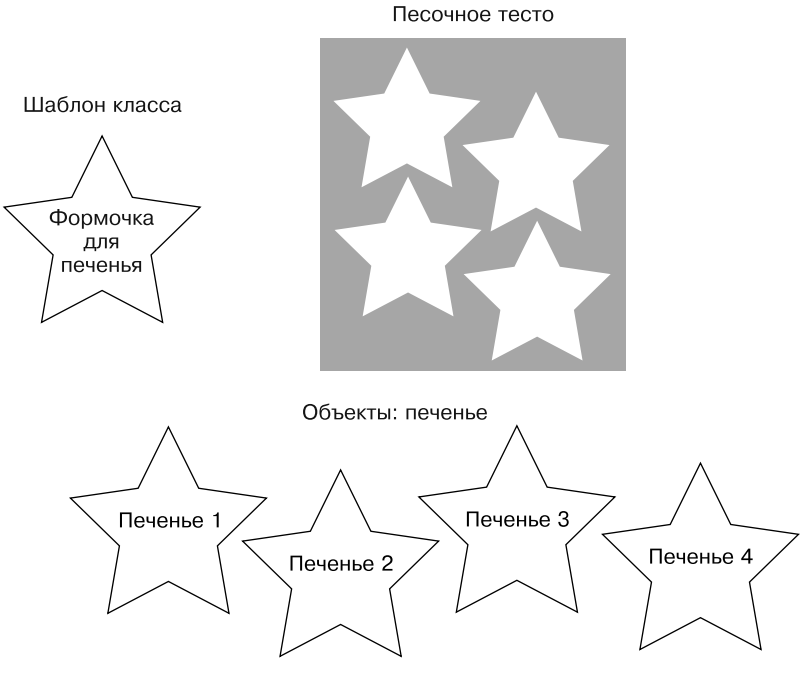

Рис. 1.10. Шаблон класса

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.

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();
  p.setName("Joe");

      ...код

  name = 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

Со временем разрабатывается все больше средств и методик 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.

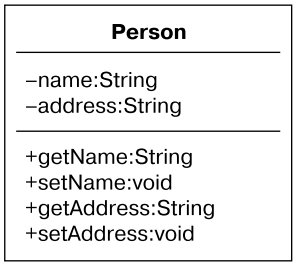

Рис. 1.11. Диаграмма класса Person

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 сокрытие 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 согласно принципу необходимого знания.

Инкапсуляция определяется тем, что объекты содержат как атрибуты, так и поведения. Сокрытие 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 сокрытие 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 сокрытия 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 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).

В целях безопасности не нужно, чтобы неконтролируемый код мог изменять или обращаться к закрытым данным. Например, во время использования банкомата нужно ввести пин-код, чтобы получить доступ к данным.

ПОДПИСИ: ИНТЕРФЕЙСЫ В СОПОСТАВЛЕНИИ С ИНТЕРФЕЙСАМИ

Не стоит путать интерфейсы для расширения классов с интерфейсами классов. Мне нравится обобщать интерфейсы, представленные методами, словом «подписи».

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 `private` 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), не повлияв 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 — системой электроснабжения). Для тостера не имеет значения, что фактической реализацией является электростанция, работающая на угле. На самом деле электричество, которое для него важно, может вырабатываться как огромной атомной электростанцией, так и небольшим электрогенератором. При такой модели любой прибор сможет получить электричество, если он соответствует спецификации интерфейса, как показано на рис. 1.12.

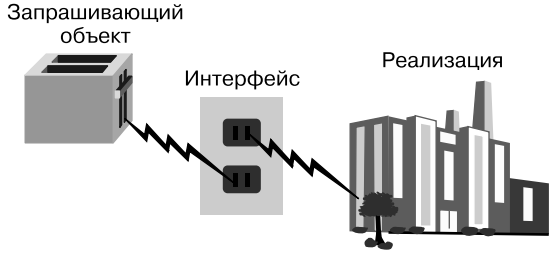

Рис. 1.12. Пример с электростанцией

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 плюс (+) обозначает `public`, а знак минус (-) указывает на `private`. Таким образом, вы сможете идентифицировать интерфейс по методам, в начале которых стоит плюс.

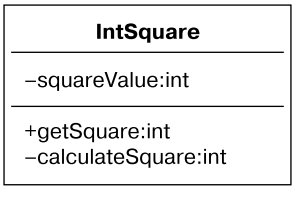

Рис. 1.13. Класс IntSquare

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

Наследование позволяет классу перенимать атрибуты и методы другого класса. Это дает возможность создавать новые классы абстрагированием из общих атрибутов и поведений других классов.

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.

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
```

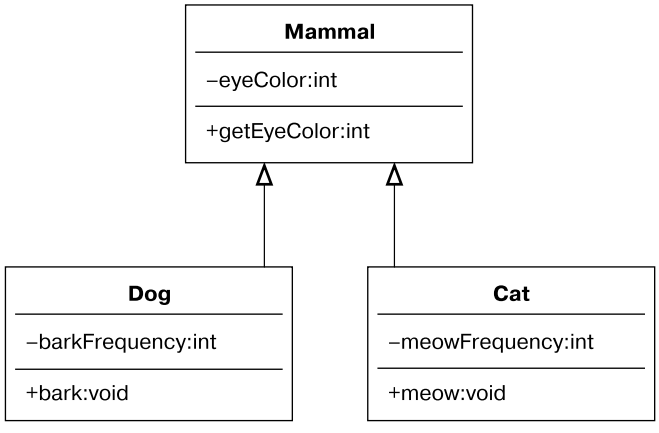

Рис. 1.14. Иерархия классов млекопитающих

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`.

ПОВЕДЕНИЕ

Стоит отметить, что сегодня поведение, как правило, описывается в интерфейсах и что наследование атрибутов является наиболее распространенным использованием прямого наследования. Таким образом, поведение абстрагируется от своих данных.

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, 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.

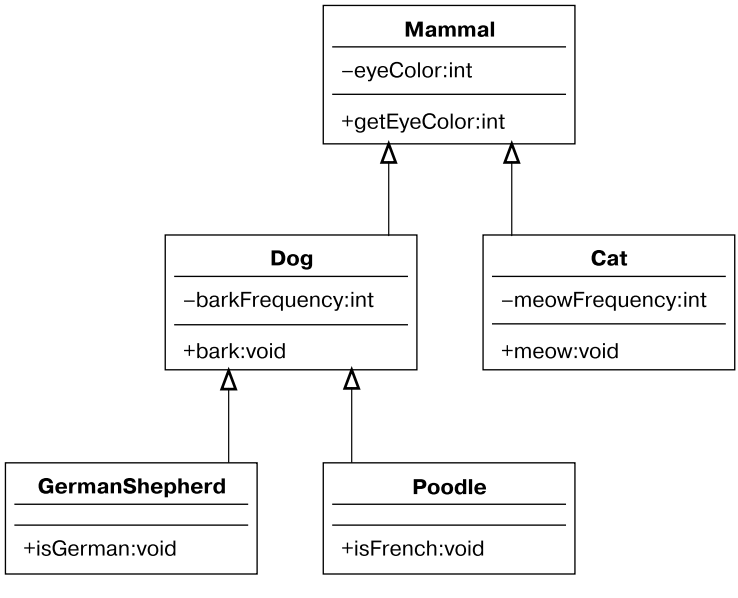

Рис. 1.15. UML-диаграмма классов млекопитающих

Такое большое количество уровней наследования является одной из причин, почему многие разработчики стараются в принципе не применять наследование. Как часто можно увидеть, непросто определить необходимую степень абстра-

гирования. Например, представим, что есть классы `penguin` (пингвин) и `hawk` (ястреб). И пингвин, и ястреб — птицы, но должны ли они оба перенимать все признаки класса `Bird` (птица), в который заложен метод умения летать?

В большинстве современных объектно-ориентированных языков программирования (например, Java, .NET и Swift) 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*.

МНОЖЕСТВЕННОЕ НАСЛЕДОВАНИЕ

Представьте себе ребенка, наследующего черты своих родителей. Какого цвета у него будут глаза? То же самое делает написание компиляторов весьма трудным. Вот C++ позволяет применять множественное наследование, а многие другие языки — нет.

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).

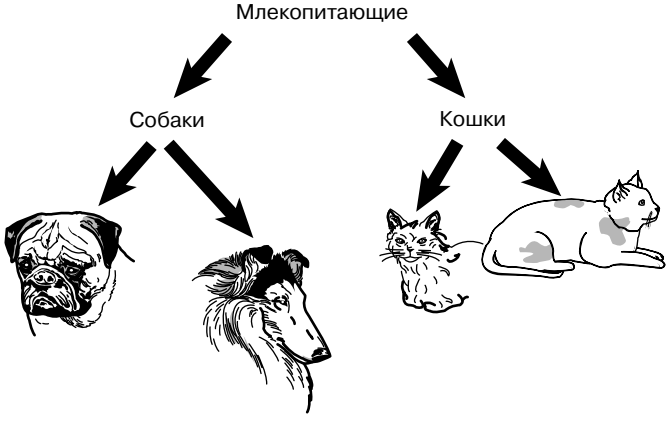

Рис. 1.16. Иерархия млекопитающих

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, какую форму она будет иметь. 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.

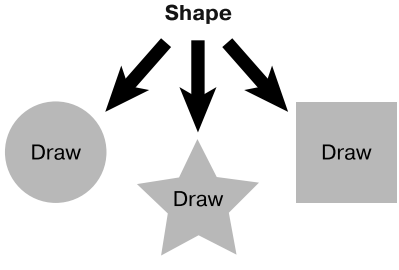

Рис. 1.17. Иерархия Shape

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.

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);
```

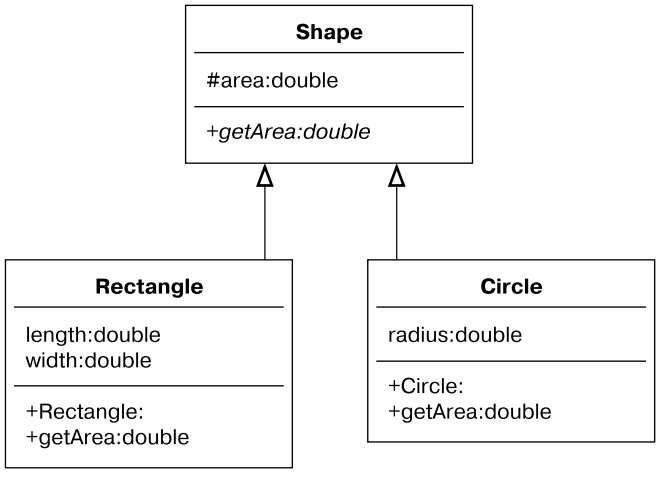

Рис. 1.18. UML-диаграмма Shape

ЧТО ТАКОЕ СТЕК?

Стек — это структура данных, представляющая собой систему «последним пришел — 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 будет последней из тех, что 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 заключается 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.

ПОЛИМОРФИЗМ С КОМПОЗИЦИЕЙ

В так называемом классическом объектно-ориентированном программировании полиморфизм традиционно выполняется наследованием. Но есть и способ выполнить полиморфизм с применением композиции. Мы обсудим такой случай в главе 12 «Принципы объектно-ориентированного проектирования SOLID».

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. Я сказал бы, что фактически 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.

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 или каком-то другом. 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, 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 сокрытия данных, присущий инкапсуляции, представляет собой инструмент, позволяющий скрывать от пользователей несущественные данные.

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

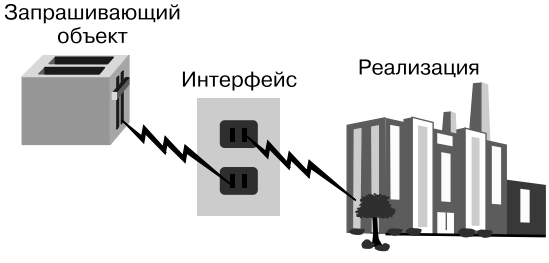

Рис. 2.1. Повторно приведенный пример с электростанцией

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.

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.

Отметим, что заменяемые двигатели должны быть идентичны, насколько это возможно во всех отношениях. Замена четырехцилиндрового двигателя на восьмицилиндровый изменит правила и, вероятно, не будет работать с другими компонентами, которые взаимодействуют с двигателем, точно так же как изменение тока с переменного на постоянный будет влиять на правила в примере силовой установки.

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 потери мощности, в действительности будет изменением интерфейса.

ЧТО ВИДЯТ ПОЛЬЗОВАТЕЛИ

Когда в этой 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 этого условия заключается суть соответствующей задачи проектирования.

КАЧЕСТВО ИНТЕРФЕЙСА

Если интерфейс спроектирован правильно, то изменения в реализации не должны требовать изменений пользовательского кода.

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, то это не изменит способ, 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 курсора.

Учитывая все эти требования, можно 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`.

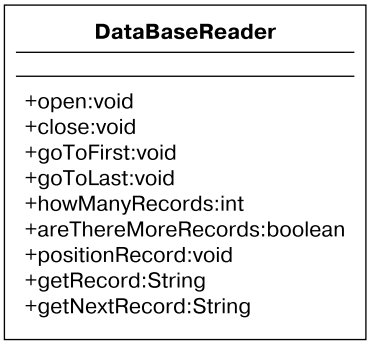

Рис. 2.2. UML-диаграмма класса DataBaseReader

Обратите внимание, что методы в этом классе открытые (помните, что рядом с именем методов, являющихся открытыми интерфейсами, присутствует знак плюс). 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 — итеративный процесс, поэтому вам необязательно 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 знать о том, как внутренний код базы данных открывает ее?
- Требуется ли знать о том, как внутренний код базы данных физически выбирает определенную запись?
- Нужно ли знать о том, как внутренний код базы определяет то, остались ли еще записи?

Ответом на все эти вопросы будет громогласное «нет»! Вам не нужно знать что-либо из этой информации. 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 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 верное 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`. Объект `Name`, который 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 самому 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.

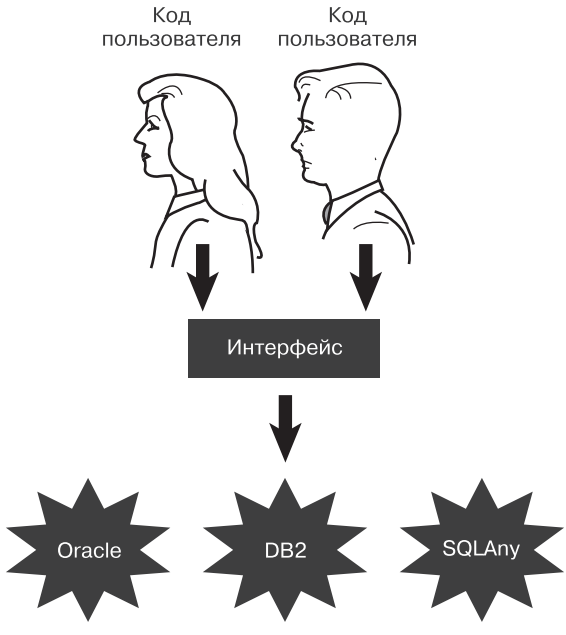

Рис. 2.3. Интерфейс

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.

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.

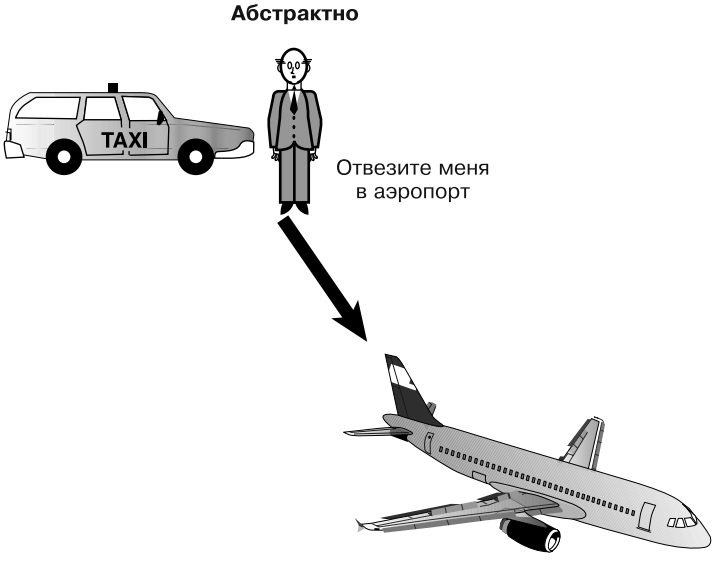

Рис. 2.4. Абстрактный интерфейс

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, 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).

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

Рис. 2.5. Не такой абстрактный интерфейс

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.

Обеспечение минимального интерфейса пользователя

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 минимального 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, и они, 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 окажется правильным 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.

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.

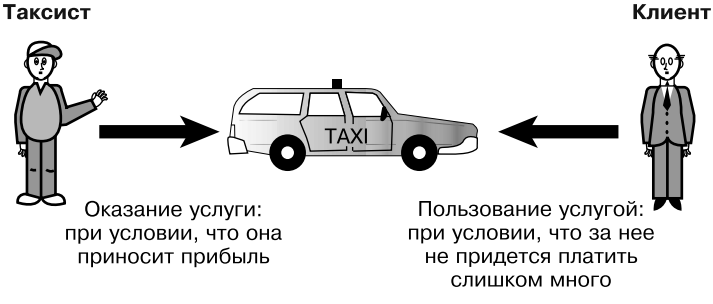

Рис. 2.6. Оказание услуги

ЗАБЕГАЯ ВПЕРЕД

Таксист, скорее всего, тоже будет объектом.

Поведения объектов

Определение пользователей — это лишь часть того, что нужно сделать. После того как пользователи будет 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 книге «Объектно-ориентированное проектирование на 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.

- Сесть в такси.
- Сказать таксисту о том, куда вас отвезти.
- Заплатить таксисту.
- Дать таксисту «на чай».
- Выйти из такси.

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

Располагать местом, до которого нужно добраться.

Подозвать такси.

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`.

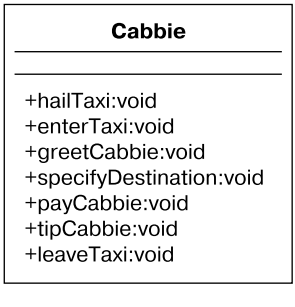

Рис. 2.7. Методы в классе Cabbie

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 доступ. 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 состояния в другое, и этих состояний много. Например, объект `DataBaseReader` будет пребывать в одном состоянии, если база данных окажется открыта, и в другом состоянии — если будет закрыта. Способ, которым все это будет представлено, зависит от того, как окажется спроектирован соответствующий класс.

Ссылки

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

Глава 3

ПРОЧИЕ 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. Хотя конструкторы обычно не используются в не объектно-ориентированных языках, таких как COBOL, C и Basic, структура, которая является частью C/C ++, действительно включает конструкторы. В первых двух главах мы упоминали об этих специальных методах, которые используются для конструирования объектов. В некоторых объектно-ориентированных языках, таких как Java и C #, конструкторы являются методами, которые имеют то же имя, что и класс. В Visual Basic .NET используется обозначение `New`, а в Swift — `init`. Как обычно, мы сосредоточимся на концепциях конструкторов, а не на конкретном синтаксисе всех языков. Давайте посмотрим на некоторый код Java, который реализует 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 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. Но все же не стоит полагаться на системные настройки по умолчанию.

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 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 или нет. Если вы не предусмотрите конструктор, 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 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 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. Это может действительно привести к необходимости правки кода, который предусматривал наличие конструктора по умолчанию.

Нужно навсегда запомнить, что 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 применяться.

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).

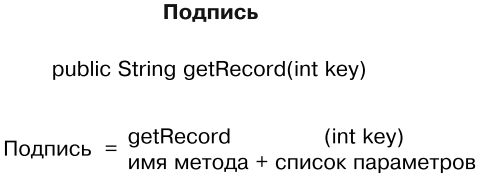

Рис. 3.1. Компоненты подписи

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 курсора в начале базы данных.
- Передать имя базы данных и позицию в базе данных, где, как мы хотим, должен установиться курсор.

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.

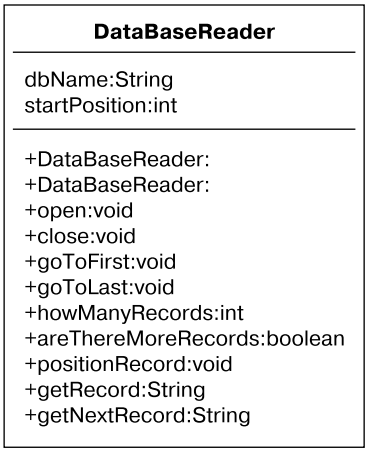

Рис. 3.2. Диаграмма класса DataBaseReader

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.

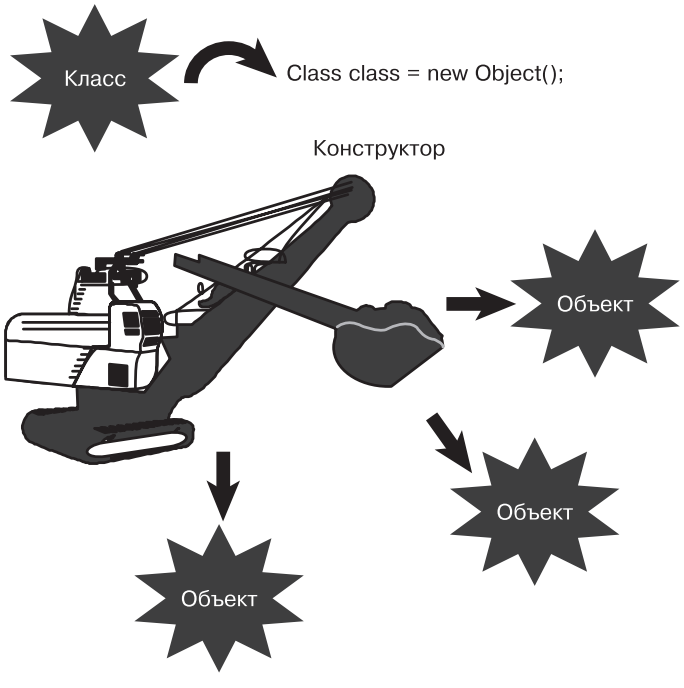

Рис. 3.3. Создание нового объекта

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

```
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, 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).

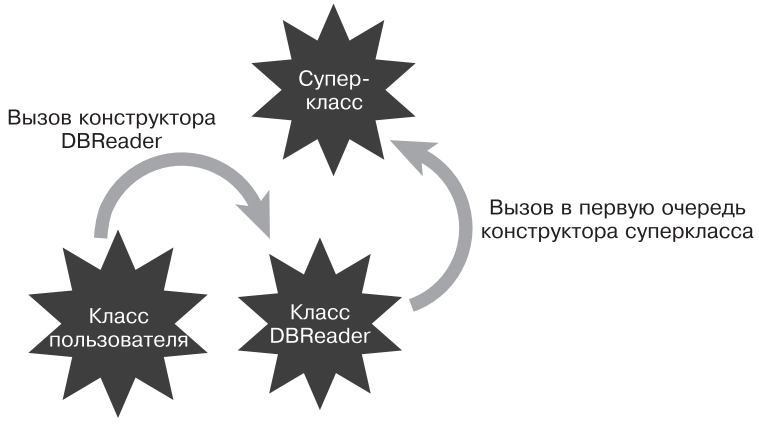

Рис. 3.4. Конструирование объекта

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 вну-

три 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.

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 ноль — 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 под-

робно 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 проверка на предмет проблем

и прерывание выполнения 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, то 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` необязательно поможет, поскольку результат может оказаться неверным. Оптимальное решение состоит в том, чтобы предложить пользователю заново ввести правильное входное значение.

СМЕШЕНИЕ МЕТОДИК ОБРАБОТКИ ОШИБОК

Несмотря на тот факт, что такая обработка ошибок необязательно будет объектно-ориентированной по своей природе, я считаю, что у нее есть законное место в объектно-ориентированном проектировании. 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 рассмотренные ранее методики выявления ошибок предпочтительнее бездействия, у них все же есть несколько недостатков. Не всегда легко определить, где именно впервые возникла проблема. Кроме того, на выявление проблемы может потребоваться некоторое время. Подробное объяснение обработки ошибок выходит за области интересов 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 будут 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 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.

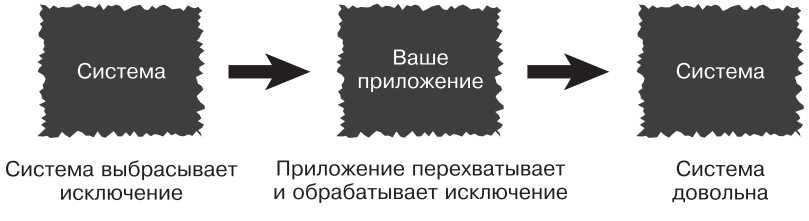

Рис. 3.5. Перехват исключения

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;
    }

}
```

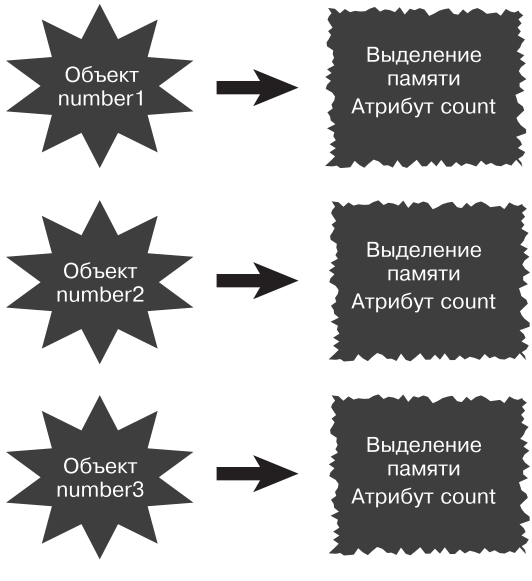

Рис. 3.6. Атрибуты объектов

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 поэкспериментировать 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 теле 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 настолько близко к глобальным данным, насколько представляется возможным при объектно-ориентированном проектировании.

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` за-

пишет данные о первой овце, информация, сохраненная `Count1`, будет потеряна. На практике статический метод не должен часто применяться. Только если вы уверены в том, что он необходим в проектировании.

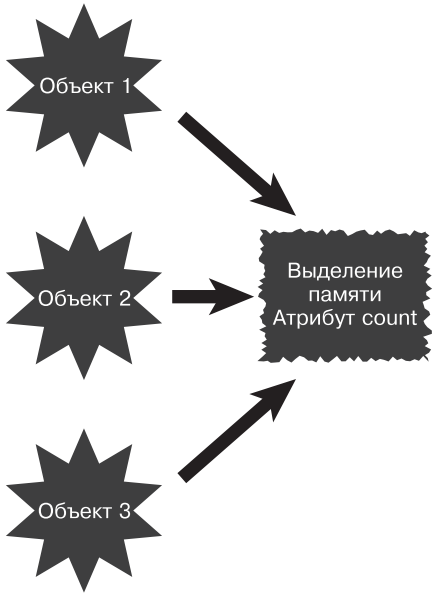

Рис. 3.7. Атрибуты классов

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 видят знак плюс, 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 плюс 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 плюс 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 плюс 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 плюс 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 плюс 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 и .NET, не позволяют выполнять перегрузку операторов.

Несмотря на это, они сами перегружают знак плюс 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.

Современная концепция наследования предполагает наследование атрибутов от одного родительского класса (простое наследование). Хотя и возможно применение множественных интерфейсов или протоколов, это нельзя назвать подлинным множественным наследованием.

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 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».

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

Рис. 3.8. Следование объектным ссылкам

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.

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

- Стивен Гилберт и Билл Маккарти, «Объектно-ориентированное проектирование на Java» (Object-Oriented Design in Java). — Беркли, штат Калифорния: The Waite Group Press (Pearson Education), 1998.
- Скотт Майерс, «Эффективное использование C++» (Effective C++). — 3-е изд. — Бостон, штат Массачусетс: Addison-Wesley Professional, 2005.
- Пол Тима, Габриэл Торок и Трой Даунинг, «Учебник по Java для начинающих» (Java Primer Plus). — Беркли, штат Калифорния: The Waite Group, 1996.

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». Другими словами, 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, описанным 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 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 получится скомпилировать.

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 {

}
```

ИСПОЛЬЗОВАНИЕ JAVA-СИНТАКСИСА

Помните, что в этой книге мы используем Java-синтаксис. Он будет в чем-то похожим, но все равно в разной мере отличаться в других языках объектно-ориентированного программирования.

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 и прочих подобных ему языках распространены комментарии 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. В C# синтаксис для комментария в документации — `///`, почти такой же применяется в документации Javadoc — `/** */`.

Первый комментарий оформлен в старомодном стиле языка программирования C и предполагает использование `/*` (слеш — звездочка) для открытия комментария и `*/` (звездочка — слеш) 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
*/
```

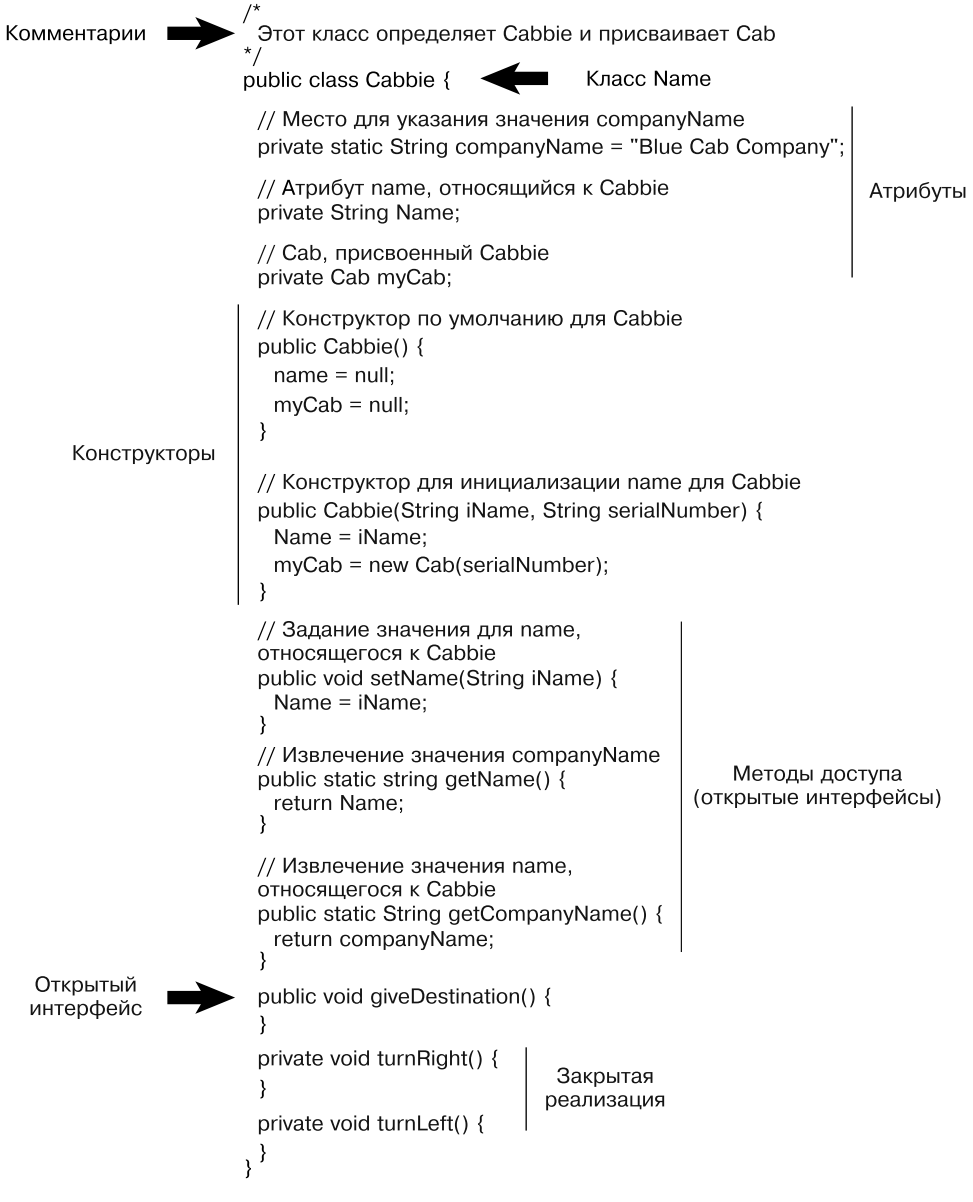

Рис. 4.1. Класс, используемый в качестве примера

Второй тип комментария предполагает использование // (слеш — слеш). При этом все, что располагается после двойного слеша 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.

СОКРЫТИЕ 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).

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.

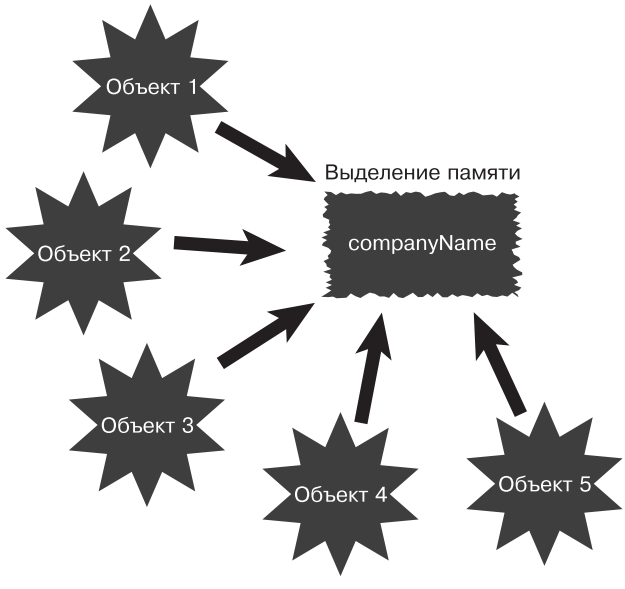

Рис. 4.2. Выделение памяти для объектов

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. Напомним, что компилятор обеспечит конструктор по умолчанию, если вы не предусмотрите конструктор 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.

БЕЗ КОНСТРУКТОРА

Написание кода без собственного конструктора вызовет работу конструктора по умолчанию. Использование же конструктора по умолчанию чревато головной болью в дальнейшем при сопровождении кода. Если код опирается на конструктор по умолчанию, но позже такой конструктор будет заменен, нужного конструктора не окажется.

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 эзотерическим, однако присвоение атрибутам значения `null` — рациональная методика программирования. Проверка той или иной переменной на предмет `null` позволяет выяснить, было ли значение должным образом инициализировано. Например, 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, вы 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` (супруг(а), 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`.

НЕСКОЛЬКО КОНСТРУКТОРОВ

Стоит отметить, что в наши дни задействование более одного конструктора считается дурной практикой. Поскольку преобладают IoC-контейнеры и им подобные, оно не поощряется, и даже не поддерживается, во многих фреймворках без особой конфигурации.

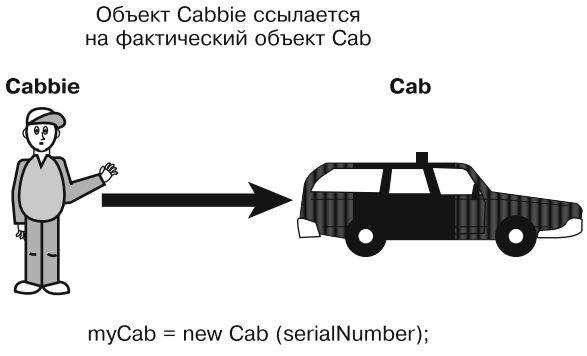

Рис. 4.3. Объект Cabbie, ссылающийся на объект Cab

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 необязательно 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.

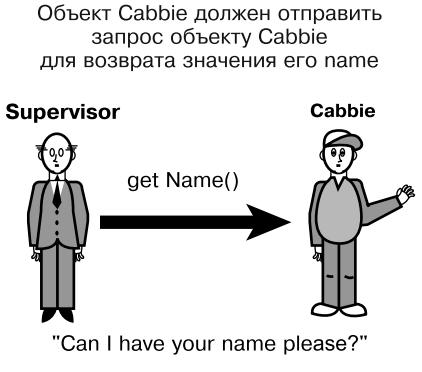

Рис. 4.4. Запрос информации

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.

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.

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.

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

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.

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 неважно, 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. 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

- Мартин Фаулер, «UML. Основы» (UML Distilled). — 3-е изд. — Бостон, штат Массачусетс: Addison-Wesley Professional, 2003.
- Стивен Гилберт и Билл Маккарти, «Объектно-ориентированное проектирование на Java» (Object-Oriented Design in Java). — Беркли, штат Калифорния: The Waite Group Press (Pearson Education), 1998.
- Пол Тима, Габриэл Торок и Трой Даунинг, «Учебник по Java для начинающих» (Java Primer Plus). — Беркли, штат Калифорния: The Waite Group, 1996.

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.

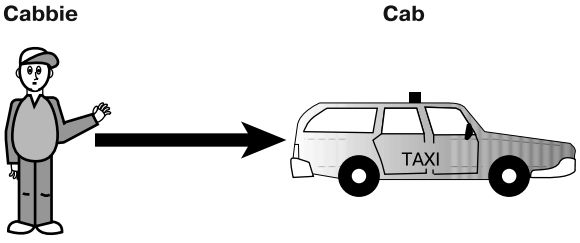

Рис. 5.1. Cabbie и Cab — реальные объекты

Когда объектно-ориентированная разработка только набирала популярность, многие структурные программисты испытывали затруднения при ее применении. Структурные программисты делали одну и ту же ошибку — создавали классы с поведениями, но без данных, в итоге получив набор из функций и подпрограмм по аналогии со структурной моделью. Это нам не нужно, поскольку в таком случае не используются преимущества инкапсуляции.

В наши дни это справедливо лишь отчасти, потому что часто разработка ведется с применением модели слабых доменов (anemic domain models), также известных как объекты передачи данных (DTO) и модели представления (view models), в которых содержится достаточно данных для заполнения представления или точно необходимое количество данных, требуемых потребителю. Гораздо больше внимания уделяется поведениям и методам обращения с данными и тому, что обрабатывается интерфейсами. Инкапсуляция поведений в интерфейсы, спроектированные по принципу единственной ответственности, и программирование интерфейсов позволяют сохранять гибкость и модульность, а также облегчают сопровождение.

ПРИМЕЧАНИЕ

Одной из моих любимых книг, содержащих указания и рекомендации по проектированию, является «Эффективное использование C++: 50 рекомендаций по улучшению ваших программ и проектов» Скотта Майерса (Effective C++: 50 Specific Ways to Improve Your Programs and Designs). 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. Как я уже отмечал ранее, на 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. Таким образом, возможно, придется уделять больше внимания

вопросам проектирования ОО в таких языках, как C ++, в отличие от Java, Swift 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 книге «Объектно-ориентированное проектирование на 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 атрибута name (рис. 5.2).

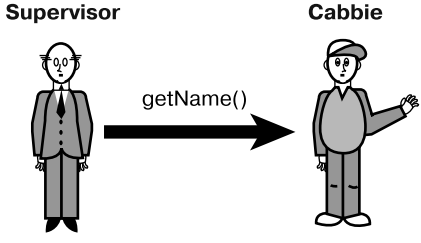

Рис. 5.2. Открытый интерфейс определяет, как объекты взаимодействуют

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.

Сокрытие 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. Реализация будет предоставлять услуги, необходимые пользователям, однако способ их предоставления не следует делать очевидным для пользователей. По своей сути изменение реализации не должно неизбежно повлечь за собой изменение в пользовательском программном коде. Опять же, лучший способ обеспечить изменение поведений — посредством интерфейсов и композиции.

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.

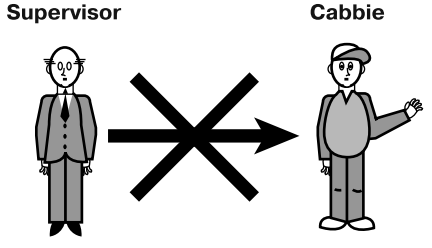

Рис. 5.3. Объектам не нужно знать некоторые детали реализации

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 «Прочие объектно-ориентированные концепции. 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.

ВНЕДРЕНИЕ КЛАССОВ КОНСТРУКТОРОМ

А вот здесь удобно будет представить концепцию внедрения классов конструктором, когда служебные классы внедряются при создании объекта (с помощью конструктора) вместо внедрения внутрь класса (с применением нового ключевого слова). Например, таксист может получить объект водительского удостоверения, объект сведений для радиосвязи (частота, позывной и т. д.), а ключ, который запускает его такси, передается в объект через конструктор.

При использовании языков программирования, в которых имеются деструкторы, жизненно важно, чтобы эти деструкторы включали соответствующие функ-

ции 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 проверке 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. Как и при грамотном подходе к проектированию классов, делайте так, чтобы документация и комментарии были понятными и все было по существу. Грамотно написанный код — сам по себе хорошая документация.

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).

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

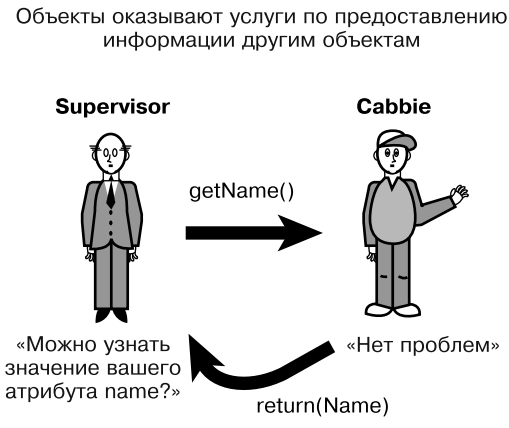

Рис. 5.4. Объекты должны запрашивать информацию

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.

Поскольку мы будем обсуждать принципы SOLID в главе 11 «Избегание зависимостей и тесно связанных классов», а также в главе 12 «Принципы объектно-ориентированного проектирования SOLID», забегая вперед, расскажу, что классы должны быть расширяемыми, но не модифицируемыми. С применением интерфейсов и их программированием у вас появится возможность использования любых паттернов, например декоратора, для расширения возможностей класса без затрагивания проверенного кода, который стал его важной частью.

КАКИЕ АТРИБУТЫ И МЕТОДЫ МОГУТ БЫТЬ СТАТИЧЕСКИМИ?

Если применяются статические методы, может происходить тесное связывание классов. Нельзя абстрагировать статический метод. Нельзя мокировать статический метод или класс. Нельзя предоставить статический интерфейс. Единственный случай, когда оправданно применение статических классов (во время разработки приложения, разработка фреймворка несколько отличается), — это когда вы имеете дело с каким-нибудь вспомогательным классом или методом расширения, который не приведет к нежелательным побочным эффектам. Например, статический класс подойдет для сложения чисел. Но статический класс плох для обращения к базам данных или веб-сервисам.

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 понимать по имени объекта, что тот собой представляет. Какие конкретно соглашения об именовании будут использоваться, в различных организациях зачастую обусловливается стандартами программирования.

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

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. Название переменных должно отражать их роль, а не их тип.

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).

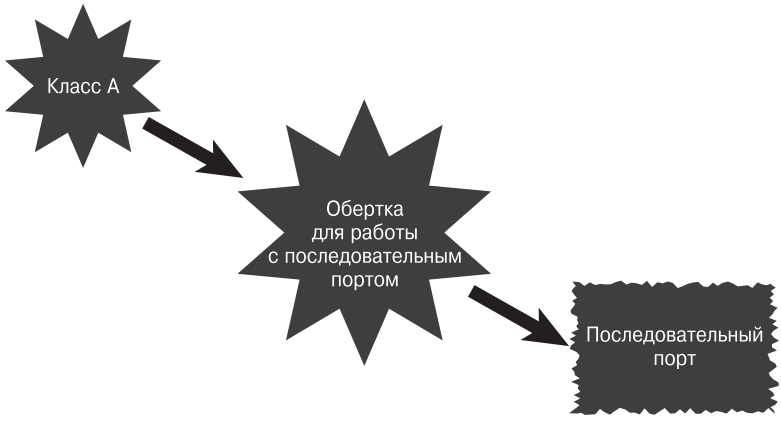

Рис. 5.5. Обертка для работы с последовательным портом

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 сокрытием 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. Применение интерфейсов хорошо помогает это обеспечить.

ОБЛАСТЬ ВИДИМОСТИ И ГЛОБАЛЬНЫЕ ДАННЫЕ

Минимизация области видимости глобальных переменных — хороший стиль программирования, но она характерна не только 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. Вы также можете обеспечить общий доступ к данным через файл или базу данных.

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.

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».

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 книге под названием «Объектно-ориентированное проектирование на 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. 5.6: 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.

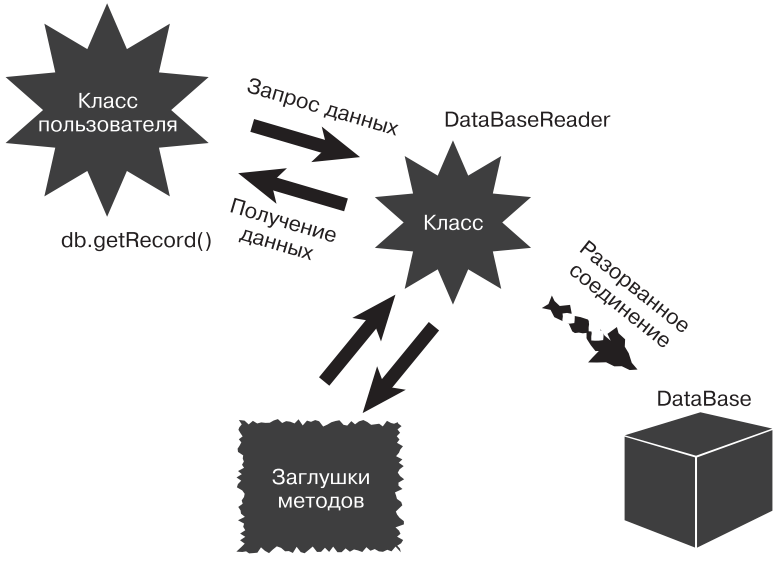

Рис. 5.6. Применение заглушек

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 booleanDBOpen = 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){
      /* DB Specific Implementation */
      return db[key];
}

   public String getNextRecord(){
      /* DB Specific Implementation */
      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, а члены команды программистов знали о них. Фактически в той или иной хорошо спроектированной программе, которую вы тестируете, заглушки должны быть интегрированы с конструкцией и сохраняться в программе для последующего использования. Коротко говоря, встраивайте функциональность для осуществления тестирования прямо в класс! Наверное, даже лучше создавайте заглушки с мок-данными, встроенные в интерфейсы, тогда вы сможете в случае надобности убрать их из самой реализации.

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 последующего 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 простой 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 типа первичных «запоминающих устройств».

- **Система простых файлов** — вы можете сохранить объект в простом файле, сериализовав его. Это определенно устаревший способ. Гораздо чаще объекты сериализуют в файлы XML и/или JSON и записывают в подобие файловой системы, массива данных или конечного устройства. Их можно разместить в базе данных или записать на диск, что в наши дни является распространенной практикой.
- **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. Это наиболее распространенный тип баз данных с гибкой структурой. Самые известные — это MongoDB и Cosmos DB.

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 простой файл или передачи его по сети.

Для передачи объекта по сети (например, файлу) система должна деконструировать этот объект (сделать его простым), 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 простой файл и извлечен позднее в том же состоянии, в каком 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.

Это еще одна причина, почему в наши дни данные отделены от поведений. Гораздо проще создать интерфейс для контракта данных и разместить его на вебсервисе, чем убедиться в том, что у обеих сторон одинаковый код.

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

- Скотт Амблер, «Введение в объектно-ориентированную технологию» (The Object Primer). — 3-е изд. — Кембридж, Соединенное Королевство: Cambridge University Press, 2004.
- Стивен Гилберт и Билл Маккарти, «Объектно-ориентированное проектирование на Java» (Object-Oriented Design in Java). — Беркли, штат Калифорния: The Waite Group Press (Pearson Education), 1998.
- Джейми Яворски, «Руководство разработчика на Java 1.1» (Java 1.1 Developers Guide). — Индианаполис, штат Индиана: Sams Publishing, 1997.
- Джейми Яворски, «Платформа Java 2 в действии» (Java 2 Platform Unleashed). — Индианаполис, штат Индиана: Sams Publishing, 1999.
- Скотт Майерс, «Эффективное использование C++» (Effective C++). — 3-е издание. — Бостон, штат Массачусетс: Addison-Wesley Professional, 2005.
- Пол Тима, Габриэл Торок и Трой Даунинг, «Учебник по Java для начинающих» (Java Primer Plus). — Беркли, штат Калифорния: The Waite Group, 1996.

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 неправиль-

ного 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 стандартный процесс 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 сосредоточиваюсь только на диаграммах классов в рамках UML. Мне нравится использовать диаграммы классов в качестве визуального средства, которое помогает при проектировании и документировании, даже если я не использую остальные доступные средства UML.

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. Я называю это «Версия 1».

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).

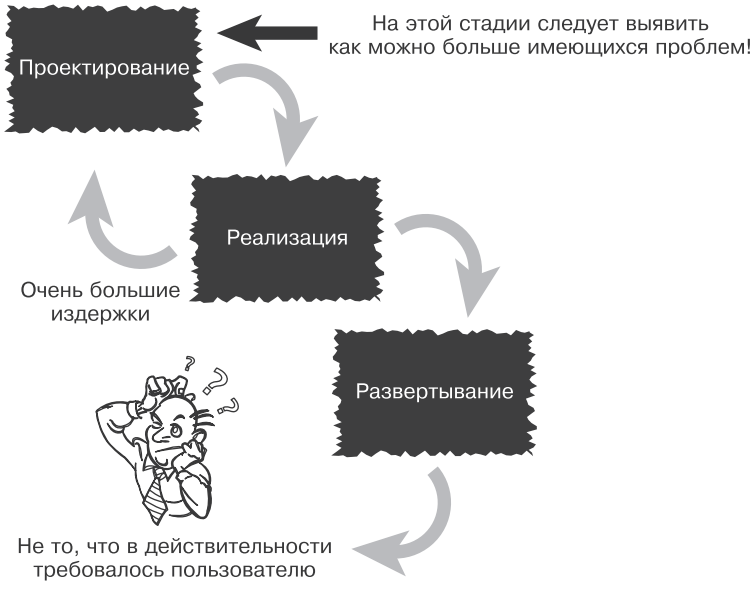

Рис. 6.1. Метод водопада

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 решите, 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 ЭКОНОМИКОЙ

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 краткосрочно ориентированная экономика 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.

Некоторые компании — производители программного обеспечения предпочитают проводить стадию бета-тестирования и тем самым дать возможность клиентам провести тестирование. Такое тестирование теоретически должно было проводиться до того, как бета-версия дойдет до клиента. Многие клиенты не так озадачены рисками при использовании предварительной версии, как стремятся получить функциональность новейшей версии. Напротив, некоторые клиенты до последнего избегают перехода на новые версии, следуя принципу «не чини то, что работает». Для них обновление — это сплошной кошмар.

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).

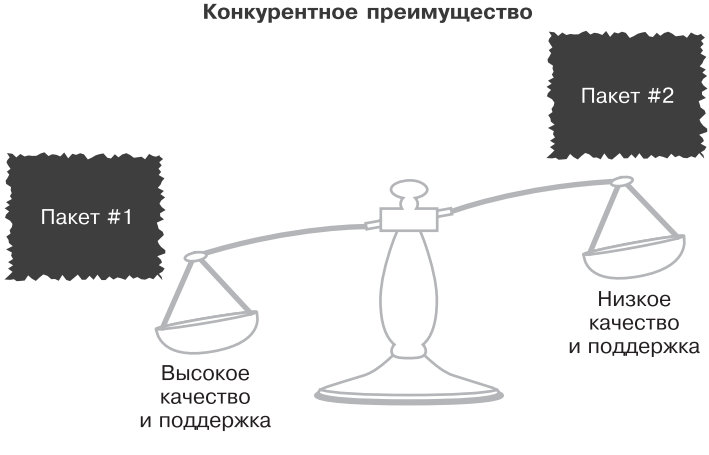

Рис. 6.2. Конкурентное преимущество

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. Необязательно 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 необязательно 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 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 на 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.

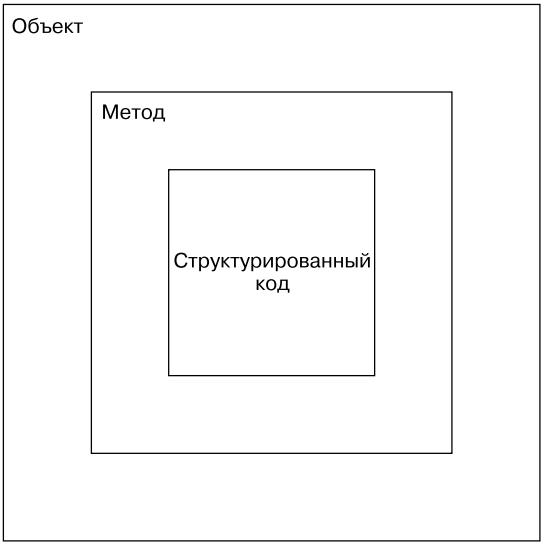

Рис. 6.3. Обертывание структурированного кода

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;

    }

}
```

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 сокрытия 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 для того, 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, то 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.

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

- Скотт Амблер, «Введение в объектно-ориентированную технологию» (The Object Primer). — 3-е изд. — Кембридж, Соединенное Королевство: Cambridge University Press, 2004.
- Стив Макконнелл, «Совершенный код: практическое руководство по разработке программного обеспечения» (Code Complete: A Practical Handbook of Software Construction). — 2-е изд. — Редмонд, штат Вашингтон: Microsoft Press, 2004.
- Стивен Гилберт и Билл Маккарти, «Объектно-ориентированное проектирование на Java» (Object-Oriented Design in Java). — Беркли, штат Калифорния: The Waite Group Press (Pearson Education), 1998.
- Джейми Яворски, «Платформа Java 2 в действии» (Java 2 Platform Unleashed). — Индианаполис, штат Индиана: Sams Publishing, 1999.
- Джейми Яворски, «Руководство разработчика на Java 1.1» (Java 1.1 Developers Guide). — Индианаполис, штат Индиана: Sams Publishing, 1997.
- Р. Вирфс-Брок, Б. Вилкерсон и Л. Вейнер, «Проектирование объектно-ориентированного программного обеспечения» (Designing Object-Oriented Software). — Аппер Сэддл Ривер, штат Нью-Джерси: Prentice-Hall, 1997.
- Мэтт Вайсфельд и Джон Киккоцци, статья «Разработка программного обеспечения рабочей группой» (Software by Committee) // Project Management, том 5, номер 1: с. 30–36, сентябрь 1999.

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), некоторые программисты вообще отказываются от наследования, выполняя проектирование только с помощью композиции.

Чаще можно встретить применение наследования интерфейсов, а не прямое наследование поведений (реализация против наследования). Наследование чаще применяется для данных/моделей, в то время как реализация — для поведений.

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 клас-

сы (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» Петера Коуда и Марка Мейфилда имеется целая глава «Проектирование с использованием композиции вместо наследования». Многие ранние объектно-ориентированные платформы даже не поддерживали истинное наследование. 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, не говоря уже о сопровождении унаследованного кода.

Да, наследование часто неправильно используют или злоупотребляют им, но это происходит из-за непонимания того, что оно собой представляет, а не потому, что использование наследования в качестве стратегии при проектировании — фундаментальный изъян.

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». В случае с интерфейсами вы можете добавить правило «ведет себя как» (implements). Данные (атрибуты), которые унаследованы, — это «является», интерфейсы, которые описывают инкапсулируемые поведения, — это «функционирует как», а композиция — это «имеет». Грани, однако, довольно размыты.

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).

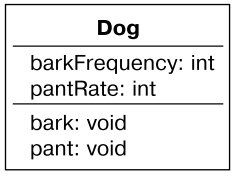

Рис. 7.1. Диаграмма класса Dog

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).

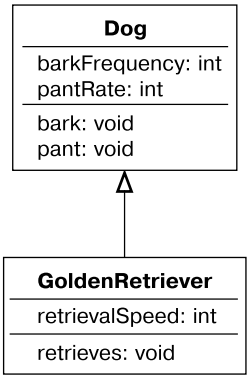

Рис. 7.2. Класс GoldenRetriever наследует от класса Dog

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).

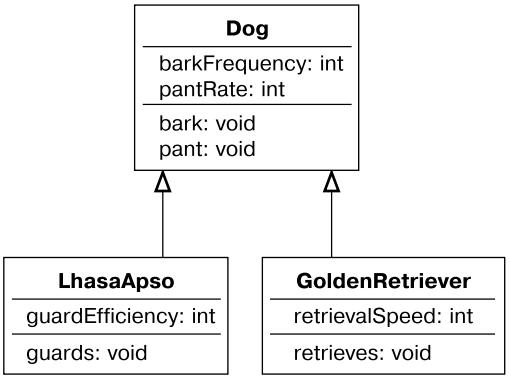

Рис. 7.3. Класс LhasaApso наследует от класса Dog

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 книге «Эффективное использование 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. Это пример одного из принципов SOLID — принципа подстановки Барбары Лисков, который мы обсудим в главе 12 «Принципы объектно-ориентированного проектирования SOLID».

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

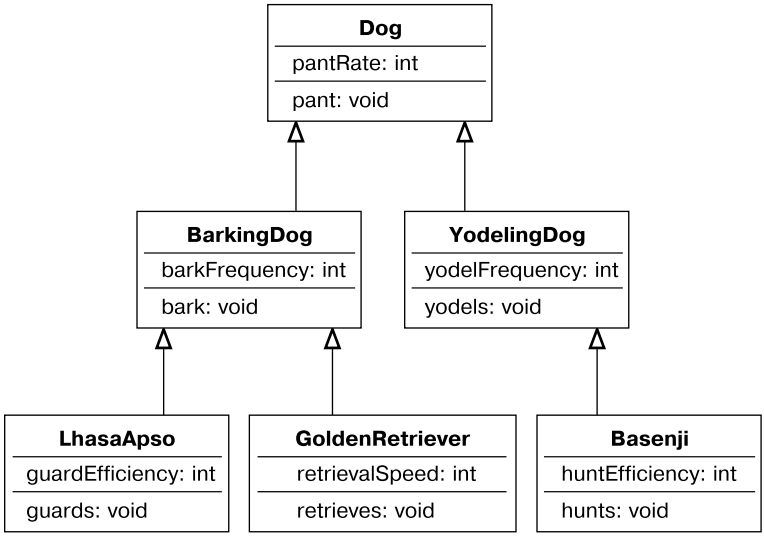

Рис. 7.4. Иерархия классов во главе с Dog

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`.

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.

Паттерн проектирования, о котором говорится в главе 10 «Паттерны проектирования», может быть в этом случае неплохим выбором. Разработчик может не создавать разновидности класса `Dog`, скорее всего, он будет либо использовать класс `Dog` (который является реализацией `IDog`), либо с помощью декоратора добавит поведения объекту `Dog`.

Проектные решения

В теории лучший подход — выделение как можно большей общности. Однако, как и во всех задачах проектирования, слишком хорошо тоже нехорошо. 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 большую 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 соб-

ственной тяжестью. Что должно случиться, если вы добавите в проект собак, издающих звуки, похожие на йодль, уже позже?

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)?

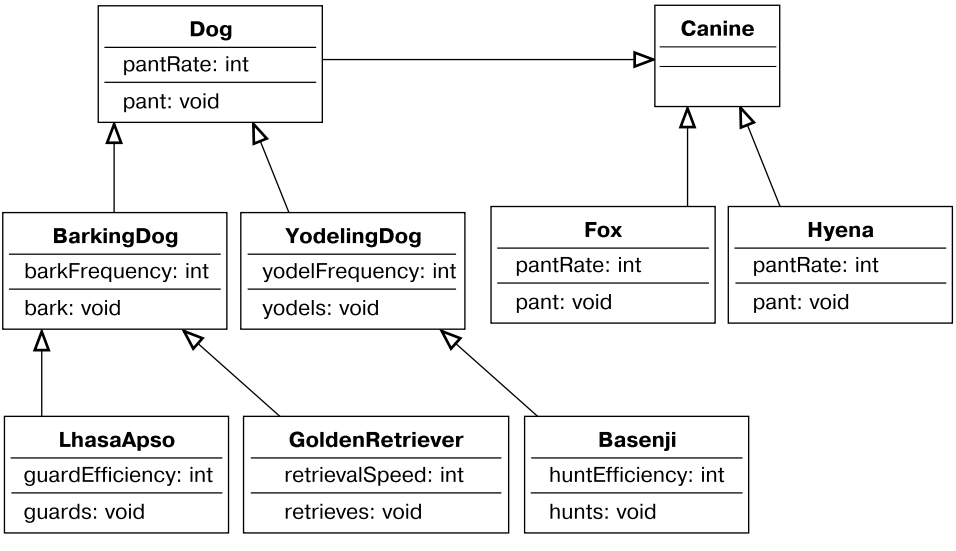

Рис. 7.5. Расширенная модель Canine

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.

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 флеш-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 на 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*.

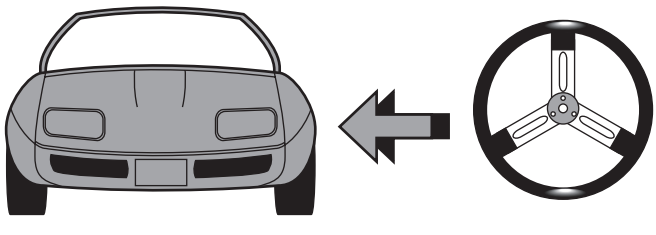

Рис. 7.6. Пример композиции

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.

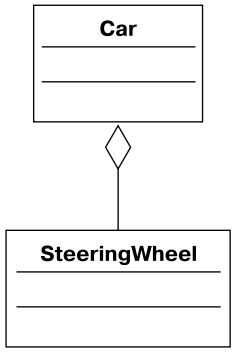

Рис. 7.7. Представление композиции на UML

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.

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 наследование. 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 «хэтчбэк». 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 `Engine`, `Stereo` и `Door`.

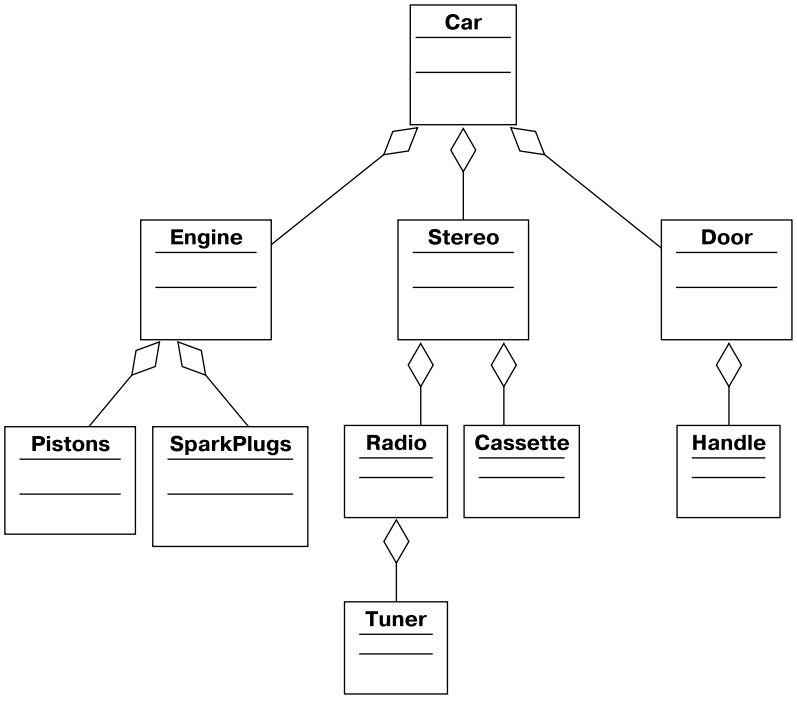

Рис. 7.8. Иерархия классов во главе с Car

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.

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.

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 сталкиваемся 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 необязательно.

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, то 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?

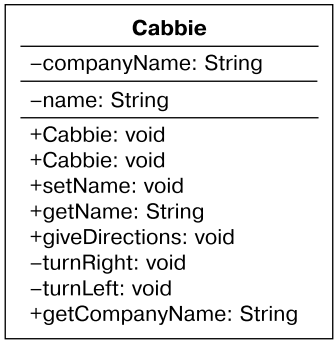

Рис. 7.9. UML-диаграмма класса Cabbie

Если бы другие классы находились в прямой зависимости от реализации класса `Cabbie`, то тестирование стало бы более сложным, а то и вовсе невозможным. С применением другого подхода при проектировании, с помощью абстрагирования поведений и наследования только атрибутов, проблемы, обозначенные выше, должны исчезнуть.

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 от-

крытую 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` необязательно инкапсулируются в классе `Cabbie`.

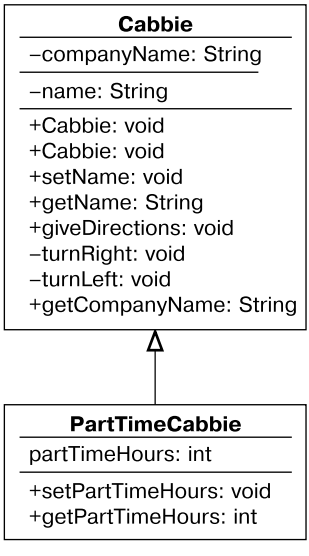

Рис. 7.10. UML-диаграмма классов Cabbie/PartTimeCabbie

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 неконкретизированная 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).

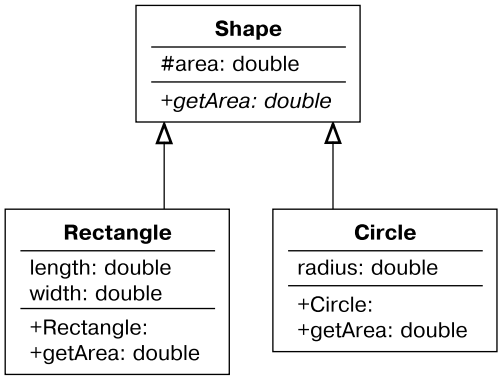

Рис. 7.11. Иерархия классов во главе с Shape

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.

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 код 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 ();

        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{

   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 и Swift не полностью реализуют 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 (Swift) 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 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

- Гради Буч, Роберт А. Максимчук, Майкл У. Энгл, Бобби Дж. Янг, Джим Коналлен и Келли А. Хьюстон, «Объектно-ориентированный анализ и проектирование с примерами приложений» (Object-Oriented Analysis and Design with Applications). — 3-е изд. — Бостон, штат Массачусетс: Addison-Wesley, 2007.

- Петер Коуд и Марк Мейфилд, «Проектирование на Java» (Java Design). — Аппер Сэддл Ривер, штат Нью-Джерси: Prentice-Hall, 1997.
- Стивен Гилберт и Билл Маккарти, «Объектно-ориентированное проектирование на Java» (Object-Oriented Design in Java). — Беркли, штат Калифорния: The Waite Group Press (Pearson Education), 1998.
- Скотт Майерс, «Эффективное использование C++» (Effective C++). — 3-е изд. — Бостон, штат Массачусетс: Addison-Wesley Professional, 2005.

Глава 8

ФРЕЙМВОРКИ И ПОВТОРНОЕ ИСПОЛЬЗОВАНИЕ: ПРОЕКТИРОВАНИЕ С ПРИМЕНЕНИЕМ ИНТЕРФЕЙСОВ И АБСТРАКТНЫХ КЛАССОВ

В главе 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 в стиле Java, протоколов 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). 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 момента написания своей первой строки 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 зависят 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»* (plug and play). 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 имеется лента, включающая разные вкладки. Эти вкладки подобны тем, что есть в презентационном пакете и программном обеспечении для работы с электронными таблицами. 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 кон-

кретной платформе, например Microsoft Windows или Linux, 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.

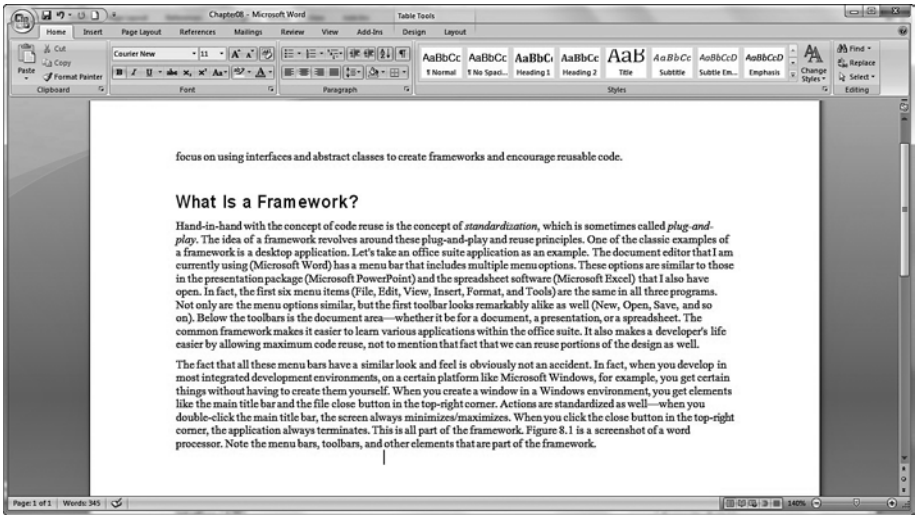

Рис. 8.1. Фреймворк для обработки текста

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 хочет-

ся, 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.

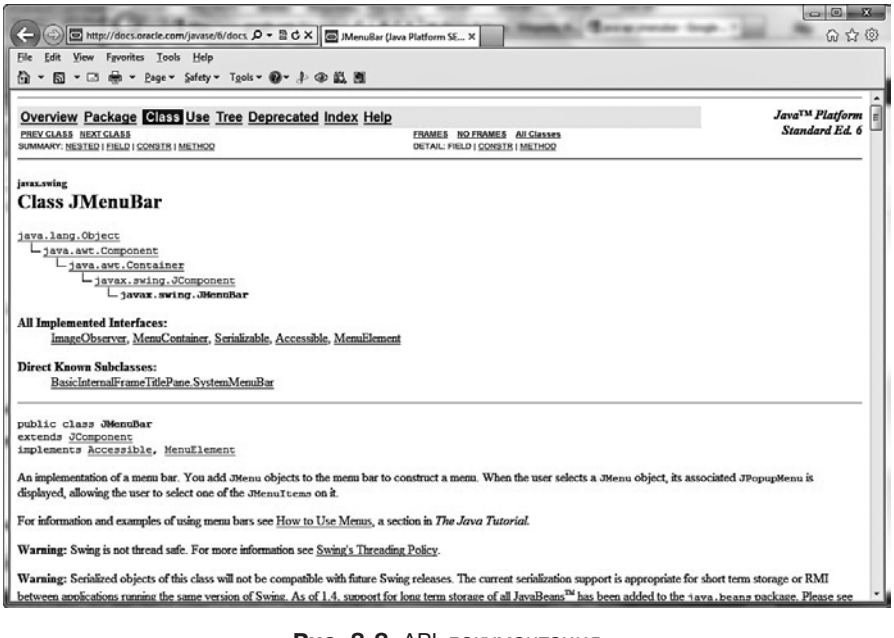

Рис. 8.2. API-документация

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.

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 Merriam-Webster (http://www.merriam-webster.com) определяет контракт как «соглашение, связывающее обязательствами двух и более человек или сторон, в частности если оно имеет законную силу».

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).

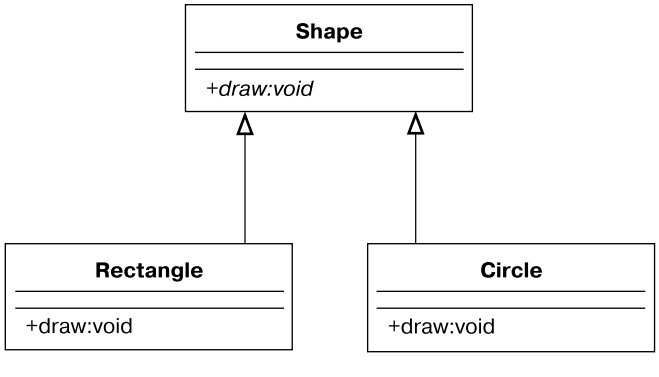

Рис. 8.3. Иерархия абстрактных классов

АБСТРАКТНЫЕ МЕТОДЫ

Обратите внимание на то, что абстрактные методы в UML-диаграммах выделены курсивом.

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 прямоугольни-
ка")};

}
```

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.

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

и Swift, абстрактные классы не предусмотрены. Таким образом, для того чтобы реализовать контракт при работе с Objective-C и Swift, вам потребуется использовать протоколы.

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 и Swift предусмотрен протокол?

Прежде всего, C++ поддерживает множественное наследование, в отличие от Java, Objective-C, Swift и .NET. Несмотря на то что классы Java, Objective-C, Swift 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.

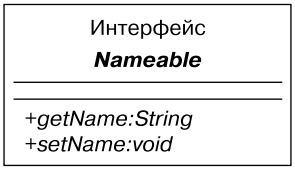

Рис. 8.4. UML-диаграмма Java-интерфейса

Обратите внимание, что `Nameable` определен на UML-диаграмме как интерфейс, благодаря чему его можно отличить от обычного класса (абстрактного или нет). Обратите внимание: 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 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).

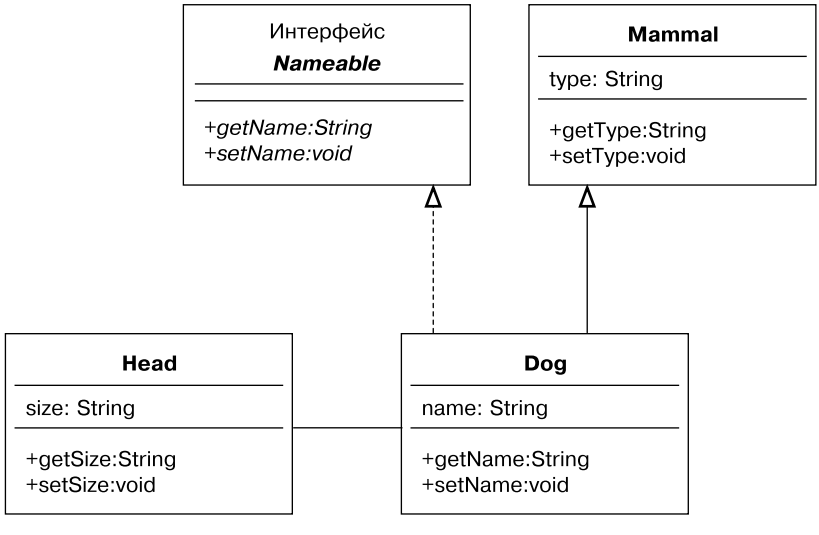

Рис. 8.5. UML-диаграмма образца кода

Если говорить кратко, то Java и .NET позволяют создавать объекты тремя путями — с помощью наследования, интерфейсов и композиции. Обратите вни-

мание, 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.

Некоторые разработчики считают, что интерфейсы — это неполноценная замена множественного наследования. Несмотря на то что интерфейсы действительно были составляющей того самого Java, который избавился от множественного наследования (что позаимствовали и многие другие языки программирования), интерфейсы и наследование применяются при проектировании в разных случаях, как нам показывает пример с `Nameable`.

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». При правильном применении отношение интерфейса больше похоже на «ведет себя как». Существуют и интерфейсы данных, которые соответствуют отношению «является экземпляром», но чаще встречаются первые.

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 в 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 (какой ужас!) 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 бы 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 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 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 3:00 захочется заказать на ужин пиццу (с салатами, хлебными палочками и напитками), которая должна быть доставлена к нему домой в 6:00. Допустим, этот клиент находится на работе (и у него, конечно же, сейчас перерыв). Он заходит на соответствующий сайт и выбирает пиццы, указывая, какого они должны быть размера, чем должны быть покрыты и должна ли у них 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 6:00. 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 будет означать громкий успех. На протяжении последующих нескольких недель клиенты Papa's Pizza будут с радостью заказывать пиццы 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 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 обес-

печения. 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 бо́льшая 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.

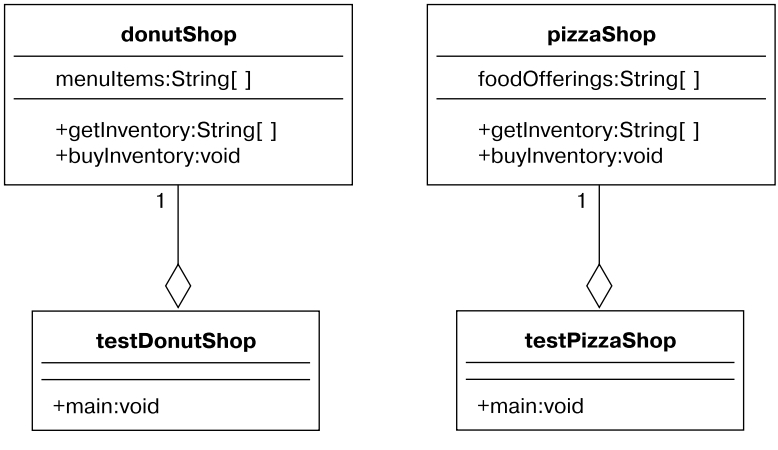

Рис. 8.6. Приложения, пути которых расходятся

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 слаженно организованных проектах такой подход

хорошо работает. Использование стандартов программирования, управление конфигурациями, управление изменениями и т. д. отлично 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 совершенно 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

Теперь 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);

}
```

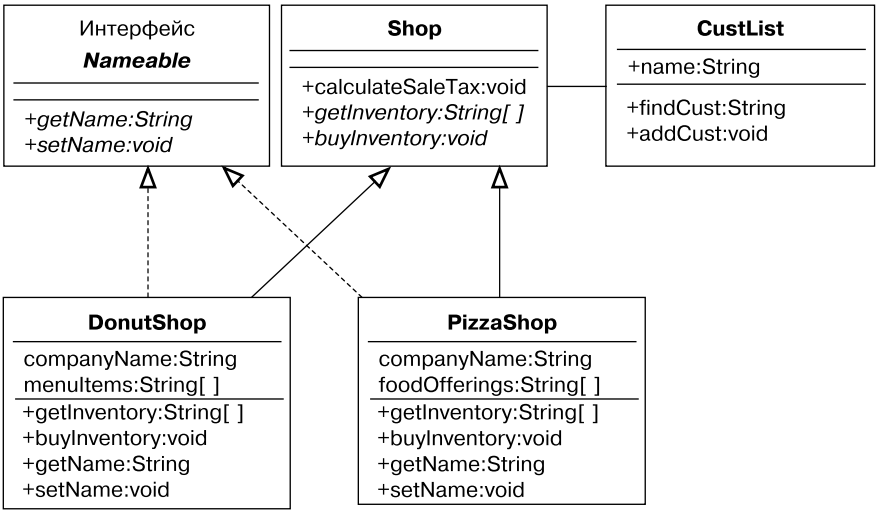

Рис. 8.7. UML-диаграмма модели Shop

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 реали-

зации 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

- Гради Буч, Роберт А. Максимчук, Майкл У. Энгл, Бобби Дж. Янг, Джим Коналлен и Келли А. Хьюстон, «Объектно-ориентированный анализ и проектирование с примерами приложений» (Object-Oriented Analysis and Design with Applications). — 3-е изд. — Бостон, штат Массачусетс: Addison-Wesley, 2007.
- Петер Коуд и Марк Мейфилд, «Проектирование на Java» (Java Design). — Аппер Сэддл Ривер, штат Нью-Джерси: Prentice-Hall, 1997.
- Скотт Майерс, «Эффективное использование C++» (Effective C++). — 3-е изд. — Бостон, штат Массачусетс: Addison-Wesley Professional, 2005.

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.

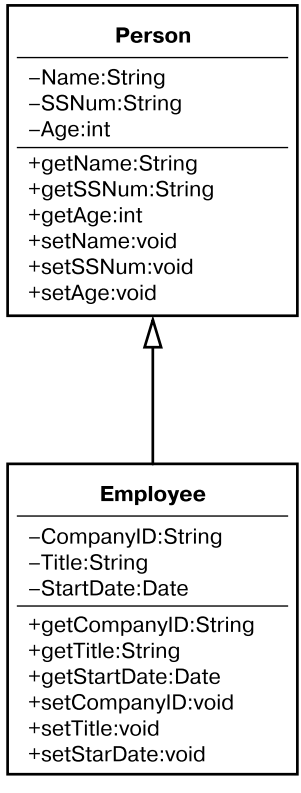

Рис. 9.1. Отношение наследования

Отношения композиции

Мы уже видели ранее: композиция означает, что тот или иной элемент является частью некого целого. Отношение наследования выражается как отношение *является экземпляром*, а композиция — как отношение *содержит как часть*. Мы интуитивно знаем, что автомобиль содержит как часть руль (рис. 9.2).

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. Мы не говорим, 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.

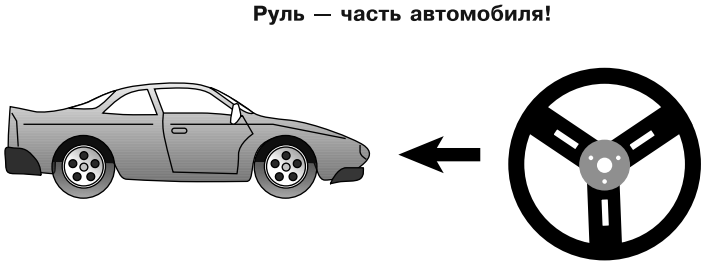

Рис. 9.2. Отношение композиции

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 «Объектно-ориентированное проектирование на 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, плеером 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.

Эта концепция становится очень важной для языков программирования вроде Java и других, включенных во 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

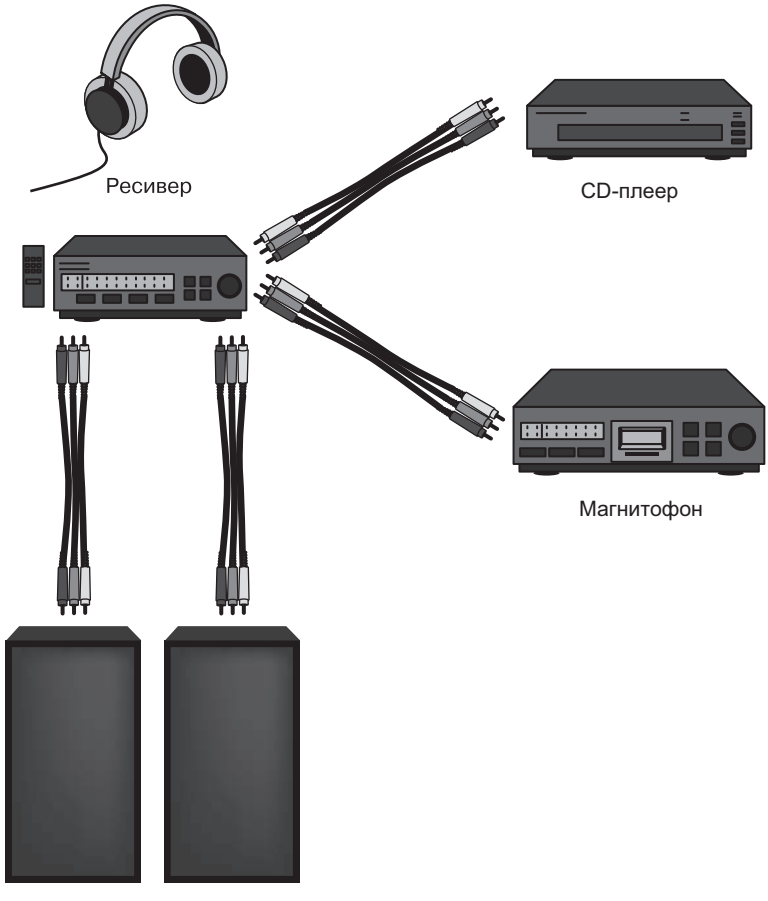

Рис. 9.3. Поэтапное создание, тестирование и верификация полной системы

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 думаете 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).

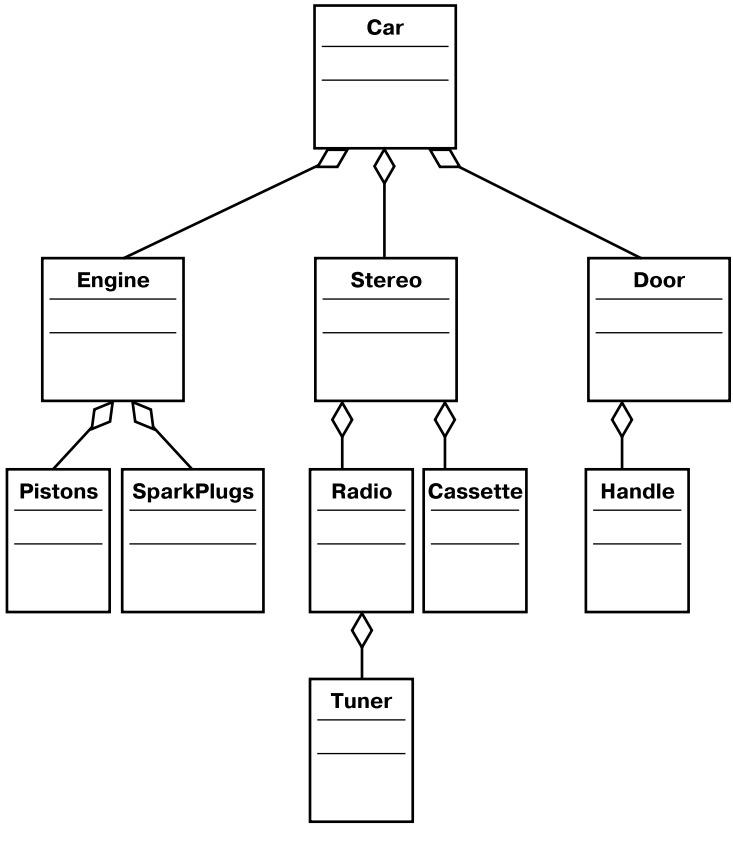

Рис. 9.4. Иерархия агрегаций для Car

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.

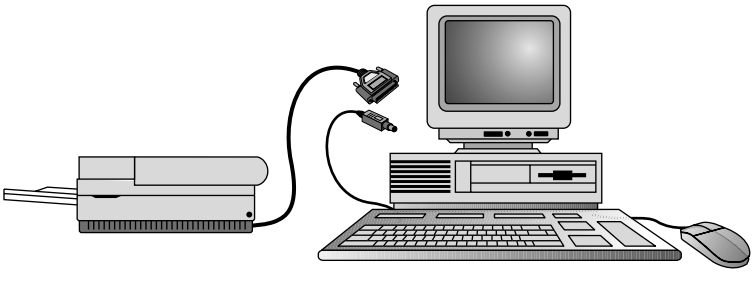

Рис. 9.5. Ассоциации как отдельная услуга

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 НЕТ

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». Стоит согласиться, что сочетание обоих устройств в одном модуле очень удобно. Но если вдруг телевизор выйдет из строя, то плеер уже не получится использовать, поскольку в этом случае он является частью одного и того же устройства.

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). Вспомните стереосистему на рис. 9.3, чтобы закрепить представление о том, что такое неинтегрированная система.

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 книге «Объектно-ориентированное проектирование на 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, 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.

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) {}

}
```

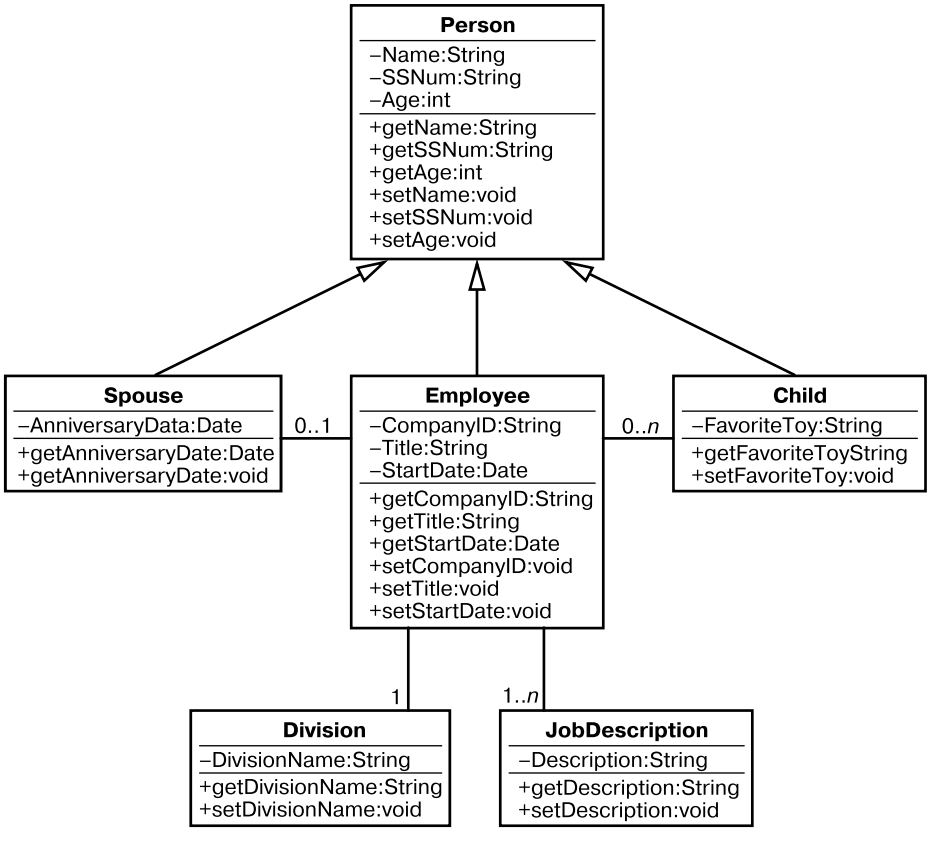

Рис. 9.7. Кардинальность на UML-диаграмме

Обратите внимание, что классы, для которых имеет место отношение «один ко многим», 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`.

Рис. 9.8. Проверка всех необязательных ассоциаций

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`.

Напомним, 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`.

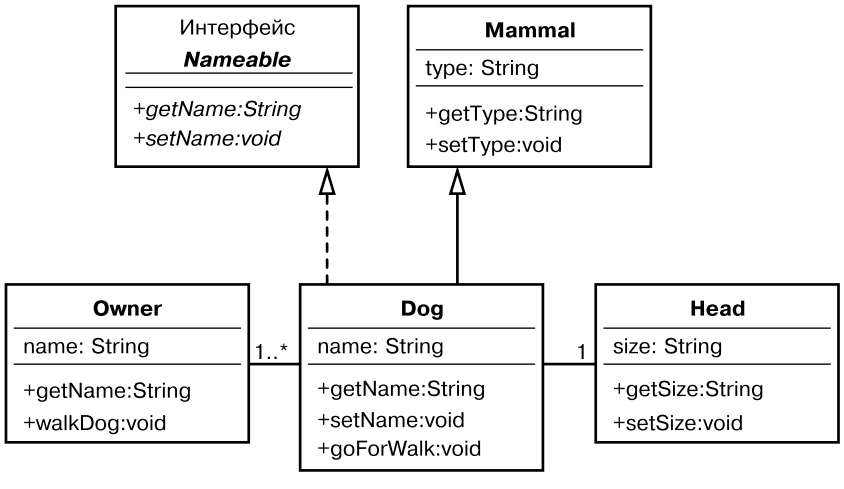

Рис. 9.9. UML-диаграмма для примера с Dog

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

- Гради Буч, Роберт А. Максимчук, Майкл У. Энгл, Бобби Дж. Янг, Джим Коналлен и Келли А. Хьюстон, «Объектно-ориентированный анализ и проектирование с примерами приложений» (Object-Oriented Analysis and Design with Applications). — 3-е изд. — Бостон, штат Массачусетс: Addison-Wesley, 2007.
- Петер Коуд и Марк Мейфилд, «Проектирование на Java» (Java Design). — Аппер Сэддл Ривер, штат Нью-Джерси: Prentice-Hall, 1997.
- Стивен Гилберт и Билл Маккарти, «Объектно-ориентированное проектирование на Java» (Object-Oriented Design in Java). — Беркли, штат Калифорния: The Waite Group Press (Pearson Education), 1998.
- Скотт Майерс, «Эффективное использование C++» (Effective C++). — 3-е изд. — Бостон, штат Массачусетс: Addison-Wesley Professional, 2005.

Глава 10

ПАТТЕРНЫ ПРОЕКТИРОВАНИЯ

В разработке программного обеспечения интересно то, что при создании системы фактически воплощается модель системы, существующей в реальном мире. К примеру, можно уверенно утверждать, что сфера информационных технологий — это бизнес или она по крайней мере дополняет его.

Чтобы написать программное обеспечение для бизнеса, разработчику требуется тщательно изучить модель работы той или иной сферы. В результате чего разработчики глубоко осведомлены о том, как проходят деловые процессы компании.

Мы уже рассматривали эту концепцию на страницах этой книги, поскольку она имеет отношение к нашему обсуждению. Например, когда мы затрагивали применение наследования для абстрагирования поведения и атрибуты млекопитающих, модель была основана на тех моделях, которые существуют в действительности, а не на модели, искусственно вымышленной для наших целей.

Получается, что когда мы создаем класс `mammal`, мы можем потом применять его для построения бесконечного количества прочих классов, например `dogs`, `cats` и т. п., потому что у всех млекопитающих есть общие поведения и атрибуты. Ради познания паттернов мы изучаем собак, кошек, белок и прочих млекопитающих. Такие паттерны позволяют нам рассмотреть животное и определить, на самом ли деле это млекопитающее или, допустим, пресмыкающееся, у которого могут быть другие паттерны поведений и атрибутов.

На протяжении всей истории люди применяли модели во многих сферах жизни, в том числе и в технической.

Эти паттерны идут рука об руку со святая святых разработки программного обеспечения — повторным использованием кода.

В этой главе мы рассматриваем паттерны проектирования, сравнительно новую область в разработке программного обеспечения (первая эпохальная книга о паттернах проектирования опубликована еще в 1995 году).

Паттерны проектирования, пожалуй, являются одним из наиболее влиятельных шагов за последние несколько лет, произошедших из движения за объектно-ориентированный подход. Сами по себе паттерны прекрасно подходят для концепции повторного использования программного кода. Из-за того, что объектно-ориентированная разработка содержит в своей основе идею повторного использования кода, объектно-ориентированная разработка идет с паттернами рука об руку.

Основная концепция паттернов проектирования вращается вокруг принципа лучших практик. Под лучшими практиками мы понимаем случаи, когда создаются подходящие эффективные решения. Такие решения документированы так, чтобы окружающие смогли получить положительные результаты из успешных действий, уже проверенных в прошлом, поскольку мы учимся методом проб и ошибок.

Одна из важнейших книг по объектно-ориентированному программированию — это «Приемы объектно-ориентированного проектирования. Паттерны проектирования», написанная Эрихом Гаммой, Ричардом Хелмом, Ральфом Джонсоном и Джоном Влиссидесом. Эта книга стала значительной вехой в сфере разработки программного обеспечения, лексика из нее распространилась в мире компьютерных наук и стала невероятно популярной, поэтому авторы книги получили прозвище *Gang of Four* (Банда четырех).

В трудах по объектно-ориентированной разработке можно часто встретить упоминание Банды четырех или просто GoF.

Цель этой главы — объяснить, что такое паттерны проектирования. Объяснение каждого паттерна проектирования выходит далеко за рамки этой книги и займет не один толстый том. Чтобы добиться цели этой главы, мы рассмотрим каждую из трех категорий паттернов проектирования (порождающий, структурный и поведенческий) по классификации Банды четырех и приведем по одному конкретному примеру из каждой категории.

Чем хороши паттерны проектирования?

Суть концепции паттернов проектирования относится не только к понятию повторного использования кода. Взначально паттерны проектирования применялись при строительстве зданий и планировании городов. Как верно подметил Кристофер Александер в работе A Pattern Language: Towns, Buildings, Construction (Язык шаблонов: города, здания, сооружения): «Каждый паттерн повторяет собой задачу, многократно возникающую в том, что нас окружает, и таким образом содержит суть решения такой задачи, тем самым позволяя повторное применение решения столько, сколько требуется, не выполняя двойной работы».

ЧЕТЫРЕ ЭЛЕМЕНТА ПАТТЕРНА

По классификации The GoF паттерн содержит четыре основных элемента:

- Название паттерна. В нем кратко выражается описание задачи проектирования, ее решений и последствий ее выполнения. Творческое мышление при создании названия для паттерна непременно расширяет словарный запас, который мы используем в проектировании. Оно помогает проектировать на более высоких уровнях абстракции. Благодаря словарному запасу, который мы используем при создании паттернов, мы можем говорить о них с нашими коллегами, вести документацию и даже самостоятельно их обдумывать. Так легче продумать конструкции и отладить их взаимодействие друг с другом. На протяжении периода разработки нашего каталога подбор подходящего названия всегда был одной из труднейших задач.
- Задача дает представление, когда применяется паттерн. Он раскрывает содержание задачи. Он может содержать особые задачи проектирования, такие как представление алгоритмов в виде объектов. Он может содержать описание структур классов и объектов, которые указывают на негибкость проектирования. Иногда задача включает в себя список условий, которые должны быть выполнены до применения паттерна, чтобы оно имело смысл.
- Решение приводит описание элементов, из которых состоит конструкция, их отношения, обязанности и взаимодействие. Решение не приводит описаний той или иной реализации конструкции, поскольку паттерн подобен лишь шаблону, который применим в различных ситуациях. Вместо этого паттерн дает абстрактное описание задачи проектирования, а также как общее расположение элементов (в нашем случае классов и объектов) эту задачу решает.
- Последствия — это результаты и компромиссы, полученные применением паттерна. Когда мы даем описание решениям в проектировании, последствия часто остаются без огласки, однако стоит заметить, что они критически важны для оценки альтернативных решений и для лучшего понимания издержек и преимуществ в результате применения паттерна. Последствия для программного обеспечения часто касаются компромиссов в пространстве и времени. Они могут также решить проблемы языка и реализации. Поскольку повторное использование кода часто является важным фактором в объектно-ориентированном проектировании, последствия применения паттерна оказывают влияние на гибкость, расширяемость и переносимость системы. Создание списка последствий поможет ясно понять и оценить их.

Схема «Модель — Представление — Контроллер» в языке Smalltalk

Чтобы понимать историческую перспективу, нужно рассмотреть схему «Модель — Представление — Контроллер» (MVC), впервые реализованную в Smalltalk (и применяемую в прочих языках объектно-ориентированного программирования). MVC часто применяется для наглядного объяснения

происхождения паттернов проектирования. Парадигма «Модель — Представление — Контроллер» применялась для создания пользовательских интерфейсов в Smalltalk. Smalltalk был, пожалуй, первым популярным объектно-ориентированным языком.

SMALLTALK

Smalltalk — это результат сочетания нескольких идей, которые зародились в стенах Xerox PARC. Эти идеи в том числе включали в себя, помимо прочих, применение мыши и графического оконного интерфейса. Smalltalk — это прекрасный язык, который заложил фундамент для дальнейшего появления объектно-ориентированных языков. Иногда разработчики отмечают, что C++ на самом деле не объектно-ориентированный, в то время как Smalltalk им является. Хотя C++ был очень популярен на заре объектно-ориентированного проектирования, Smalltalk всегда имел свой узкий круг приверженцев. Java же — это язык, охватывающий и разработчиков на C++.

Понятие паттернов проектирования распределяет роли компонентов MVC следующим образом:

> Модель отвечает за функционирование самого приложения, представление отвечает за графическое отображение, а контроллер — за то, как интерфейс реагирует на ввод данных пользователем.

Проблема предыдущих парадигм в том, что модель, представление и контроллер раньше были одной сущностью. Предположим, что какой-либо единый объект содержит все три компонента. В парадигме MVC у каждого из трех компонентов будут собственные интерфейсы, отделенные от интерфейсов других. Поэтому если требуется изменить интерфейс пользователя, то придется вносить изменения только в представление. На рис. 10.1 показана схема MVC при проектировании.

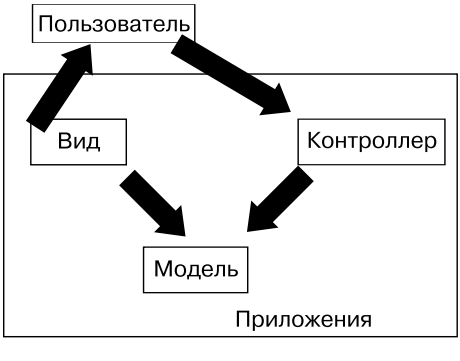

Рис. 10.1. Парадигма «Модель — Представление — Контроллер»

Помните, многое из того, что мы изучили в области объектно-ориентированной разработки, сталкивается с проблемой интерфейсов и реализации. Насколько это возможно, мы стараемся отделить интерфейс от реализации. В той же мере мы стараемся отделять интерфейсы друг от друга. Например, мы не хотим сочетания нескольких интерфейсов, которые не имеют никакого отношения друг к другу (или к решению рассматриваемой проблемы). Парадигма MVC была одной из первых в таком разделении интерфейсов.

Паттерн MVC четко определяет интерфейсы между отдельными компонентами, которые относятся к одной из основных распространенных задач в программировании — созданию пользовательских интерфейсов, их связи с бизнес-логикой и данными.

Если происходит разделение пользовательских интерфейсов, бизнес-логики и данных по паттерну MVC, получится гибкая и устойчивая система. Допустим, пользовательский интерфейс находится на клиентской машине, бизнес-логика — на сервере приложения, а данные — на сервере для них. Разработка приложения таким способом позволит изменить графический интерфейс без всякого вмешательства в бизнес-логику и данные. Подобным образом, если нужно изменить бизнес-логику и посчитать то или иное поле по-другому, можно внести в нее изменения, не затрагивая интерфейс. И наконец, если нужно заменить базу данных и хранить данные другим способом, можно сменить способ хранения данных на сервере без вмешательства как в интерфейс, так и в бизнес-логику. Это несомненно предполагает неизменность интерфейсов между тремя составляющими.

ПРИМЕР MVC

В качестве примера можно привести окно со списком в пользовательском интерфейсе. Представьте графический интерфейс, который включает в себя список телефонных номеров. Само окно — это представление, список номеров — это модель, а контроллер представляет собой логику, соединяющую окно и список.

НЕДОСТАТКИ MVC

Несмотря на то что принцип MVC великолепен, могут возникнуть трудности, поскольку много внимания требуется уделить прозрачности исполнения. Это общая задача для всего объектно-ориентированного проектирования — существует зыбкая грань между аккуратным и громоздким исполнением. Остается вопрос: с учетом оценки всего исполнения, насколько сложной будет созданная система?

Типы паттернов проектирования

Существует 23 паттерна, разбитых по группам на три категории, которые приведены ниже. Большинство примеров написано на C++, а некоторые — на Smalltalk. Для времени первого издания книги характерно применение C++

и Smalltalk. В 1995 году мир как раз находился на пороге интернет-революции и, соответственно, популярности языка программирования Java.

После того как преимущества паттернов проектирования стали очевидными, множество других книг вышло покорять появившийся рынок.

В любом случае на самом деле неважно, какой язык используется. «Приемы объектно-ориентированного проектирования. Паттерны проектирования» представляет собой руководство по проектированию, а сами паттерны находят применение в бесчисленном количестве языков программирования. Авторы книги разделили паттерны на три категории.

- Порождающие паттерны позволяют создавать объекты, благодаря этому их не приходится создавать непосредственно разработчику. Применение паттернов обеспечивает большую гибкость программы при выборе объектов, которые требуется создать в зависимости от случая.
- Структурные паттерны позволяют объединять группы объектов в более сложные конструкции, например сложные пользовательские интерфейсы или учетные данные.
- Поведенческие паттерны позволяют задать способы взаимодействия объектов в системе и управления потоком в сложной программе.

Чтобы дать понимание того, что представляют собой паттерны проектирования, в следующих разделах приводится разбор одного примера для каждой из трех категорий. В источниках, перечисленных в конце этой главы, можно найти полный список и описание отдельных паттернов проектирования.

Порождающие паттерны

Есть несколько категорий порождающих паттернов:

- абстрактная фабрика;
- строитель;
- фабричный метод;
- прототип;
- одиночка.

Как говорилось ранее, задача этой главы — дать объяснение, что такое паттерн проектирования, а не подробный разбор каждого паттерна из книги Банды четырех. Поэтому мы рассмотрим по одному паттерну из каждой категории.

С учетом вышесказанного рассмотрим пример порождающего паттерна — «фабричный метод».

Фабричный метод

Создание или обработка объектов по праву может считаться одной из концепций, на которых зиждется объектно-ориентированное программирование. Само собой разумеется, что невозможно использовать объект, покуда он не существует. При написании кода наиболее очевидный способ при создании объекта — использовать ключевое слово `new`.

Чтобы проиллюстрировать это, давайте обратимся к примеру с фигурами, которые уже встречались в этой книге. Вот уже знакомый нам родительский класс `Shape`, который абстрактен, и дочерний класс `Circle`, который представляет собой конкретную реализацию. Мы создаем экземпляр класса `Circle` обычным способом посредством ключевого слова `new`:

```
abstract class Shape {
}

class Circle extends Shape {

}

Circle circle = new Circle();
```

Хотя такой код обязательно будет работать, в нем могут находиться также многие другие места, где нужно создать экземпляр класса `Circle`, либо, в зависимости от случая, класса другой фигуры. Во многих случаях при создании объекта будут требоваться определенные параметры, которые придется задавать каждый раз при создании класса `Shape`.

В результате каждый раз при смене способа создания объектов требуется вносить изменения в код в каждом месте, где нужно создать объект `Shape`. Такой код в высокой степени связан, поскольку изменение в одном месте потенциально требует изменений также во многих других местах. Еще одна проблема этого подхода заключается в том, что он предоставляет логику создания объектов программистам при помощи классов.

Исправить ситуацию позволит применение паттерна «фабричный метод». В двух словах, «фабричный метод» отвечает за инкапсуляцию всех экземпляров, обеспечение единообразия во всей реализации.

Вы используете фабрику для создания экземпляра, а фабрика отвечает за создание экземпляра должным образом.

Паттерн «фабричный метод»

Основное назначение паттерна «фабричный метод» — создание объекта без необходимости точного указания класса, то есть использование интерфейсов для создания новых типов объектов.

Чтобы дать представление о реализации этого паттерна, в качестве примера создадим «фабрику» для класса `Shape`. На рис. 10.2 изображена диаграмма классов, которая наглядно раскрывает, как в этом примере взаимодействуют различные классы.

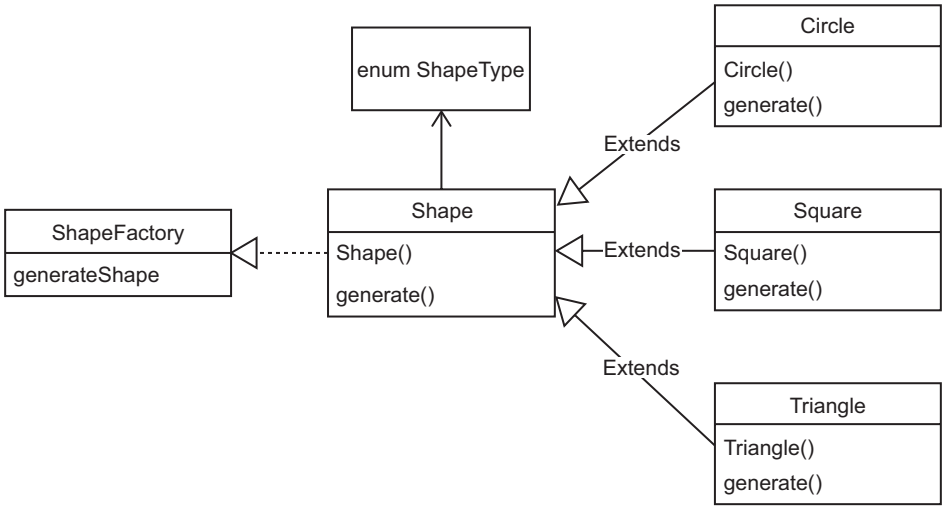

Рис. 10.2. Создание «фабрики» для класса Shape.

В некотором роде можно говорить о «фабрике» как об обертке. Стоит учитывать то, что при создании объекта может быть задействована какая-то важная логика, поэтому не нужно, чтобы программист (пользователь) мог видеть эту логику, — почти как в концепции метода мутаторов (геттеры и сеттеры), когда возврат значения находится внутри какой-либо логики (вроде случаев, когда требуется ввести пароль). Использование «фабричного метода» полезно, когда заведомо неизвестно, какой класс может понадобиться. К примеру, известно, что нужна будет фигура, но вот какая именно — никто не имеет понятия (по крайней мере, пока). Поэтому нужно не забывать, что все возможные классы должны находиться в одной иерархии, то есть все классы в этом примере будут подклассами класса `Shape`. Собственно, «фабрика» нужна именно тогда, когда точно неизвестно, что именно нужно, благодаря ей можно добавлять некоторые классы впоследствии.

Если бы было известно, что нужно конкретно, можно было бы просто вставить необходимый экземпляр с помощью конструктора или сеттера.

В своей основе это вытекает из определения полиморфизма.

Мы создаем перечисляемый тип с помощью кодового слова `enum`, который будет содержать разные типы фигур. В этом примере будут фигуры `CIRCLE` (круг), `SQUARE` (квадрат) и `TRIANGLE` (треугольник).

```
enum ShapeType {
    CIRCLE, SQUARE, TRIANGLE
}
```

Абстрактный класс Shape задается только с помощью конструктора и абстрактного метода generate ().

```
abstract class Shape {

    private ShapeType sType = null;

    public Shape(ShapeType sType {
        this.sType = sType;
    }

// Создает защищенную форму абстрактного void generate ();

}
```

Дочерние классы CIRCLE, SQUARE и TRIANGLE расширяют класс Shape, идентифицируют себя и обеспечивают конкретную реализацию метода generate().

```
class Circle extends Shape {

    Circle() {
        super (ShapeType.CIRCLE);
        generate();
     }

    @Override
    protected void generate() {
         System.out.printin("Generating a Circle");
    }
}

class Square extends Shape {

    Square() {
        super (ShapeType. SQUARE);
        generate();
    }

    @Override
    protected void generate() {
        System.out.printin("Generating a Square") ;
    }
}
class Triangle extends Shape {

    Triangle() {
        super (ShapeType. TRIANGLE) ;
        generate();
```

```
    }

    @Override
    protected void generate() {
        System.out.printin("Generating a Triangle");
    }
}
```

Класс ShapeFactory, как видно из названия, на самом деле является «фабрикой». Сосредоточимся на методе generate (). При том, что «фабрика» имеет много преимуществ, не стоит забывать, что метод generate() — это единственное место в приложении, которое действительно создает экземпляр класса Shape.

```
class ShapeFactory {
        public static Shape generateShape(ShapeType sType) {
            Shape shape = null;
            switch (sType) {

             case CIRCLE:
                 shape = new Circle();
                 break;

            case SQUARE:
                shape = new Square();
                break;

            case TRIANGLE:
                shape = new Triangle();
                break;

            default:
                // throw an exception
                break;
            }
            return shape;
    }
}
```

При традиционном подходе для создания каждого отдельного объекта программист будет вынужден собственноручно создавать объекты с помощью ключевого слова new, как указано ниже:

```
public class TestFactoryPattern {
     public static void main(String[] args) {

          Circle circle = new Circle();
          Square square = new Square();
          Triangle triangle = new Triangle();
     }
}
```

Чтобы правильно использовать «фабрику», программист должен применять класс `ShapeFactory` для получения какого-либо из объектов `Shape`:

```
public class TestFactoryPattern {
    public static void main(String[] args) {

        ShapeFactory.generateShape(ShapeType.CIRCLE) ;
        ShapeFactory.generateShape (ShapeType. SQUARE) ;
        ShapeFactory.generateShape(ShapeType. TRIANGLE) ;
    }
}
```

Структурные паттерны

Структурные паттерны используются для создания сложных структур из групп объектов. Следующие семь паттернов проектирования принадлежат к категории структурных:

- адаптер;
- мост;
- компоновщик;
- декоратор;
- фасад;
- приспособленец;
- заместитель.

В качестве примера структурного паттерна рассмотрим «адаптер». Паттерн «адаптер» также является одним из важнейших паттернов проектирования. Этот паттерн дает хорошее представление о том, как реализация отделена от интерфейса.

Паттерн «адаптер»

Паттерн «адаптер» дает возможность создать другой интерфейс для уже существующего класса. Сам по себе паттерн «адаптер» предоставляет обертку для класса. Другими словами, создается новый класс, который включает в себя (обертывает) функциональность уже существующего класса посредством нового, а по возможности — лучшего интерфейса. Простой пример обертки — это класс `Integer` в языке Java. Класс `Integer` обертывает собой одно целочисленное значение. Возникает вопрос, зачем вообще это надо. Дело в том, что в объектно-ориентированной системе все является объектом. В языке Java примитивные типы наподобие `int`, `float` и т. д. не являются объектами. Когда необходимо выполнить какие-либо действия над такими типами, например преобразования, требуется представить их как объекты. Таким образом, для этого создается объ-

ект-обертка, который содержит в себе примитивный тип. Можно взять такое значение примитивного типа:

```
int myInt = 10;
```

и обернуть его в объект `Integer`:

```
Integer myIntWrapper = new Integer (myInt);
```

Теперь можно выполнить преобразование типа посредством следующей строки:

```
String myString = myIntWrapper.toString();
```

Такая обертка дает возможность представлять исходное целое число как объект, предоставляя все преимущества объекта.

Что касается самого паттерна «адаптер», рассмотрим пример интерфейса электронной почты. Предположим, что вы приобрели какой-либо код, который предоставляет функциональность, которая необходима для обеспечения работы почтового клиента. Этот инструмент предоставляет все необходимое для почтового клиента, за исключением случаев, когда нужно внести в интерфейс небольшие правки. Фактически все, что нужно сделать, — это изменить интерфейс прикладного программирования (API) для получения почты.

Ниже приведен класс, который является простым примером почтового примера в данном контексте:

```
package MailTool;
public class MailTool {
    public MailTool () {
    }
    public int retrieveMail() {

        System.out.printin ("You've Got Mail");

            return 0;
    }
}
```

Когда происходит вызов метода `retrieveMail()`, получение нового письма сопровождается уведомлением «You've Got Mail» (у вас новое письмо). Теперь предположим, что нужно изменить интерфейс во всех клиентах компании с `retrieveMail()` на `getMail()`. Можно создать интерфейс для обеспечения этого:

```
package MailTool;
    interface MailInterface {
        int getMail();
}
```

Теперь можно создать свою почтовую службу, которая обертывает собой ту, что была изначально, и предоставляет необходимый 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 данного класса создается экземпляр исходной почтовой службы, которую необходимо усовершенствовать. Этот класс реализует интерфейс `Mail`, который принудительно задействует метод `getMail()`. Внутри этого метода буквально происходит вызов метода `retrieveMail()` той почтовой службы, которая была изначально.

Чтобы задействовать новый класс, требуется создать экземпляр новой почтовой службы и вызвать метод `getMail()`.

```
package MailTool;
public class Adapter
{
    public static void main(String[] args)
    {
        MyMailTool myMailTool = new MyMailTool();

        myMailTool.getMail();

    }
}
```

Когда происходит вызов метода `getMail()`, с помощью такого нового интерфейса происходит вызов метода `retrieveMail()` из первоначальной почтовой службы. Это очень простой пример. И все же благодаря созданию обертки можно улучшить интерфейс и добавить новый функционал к уже существующему классу.

Несмотря на то что концепция паттерна «адаптер» довольно проста, с помощью него можно создавать новые интерфейсы с широкими возможностями.

Паттерны поведения

Паттерны поведения, или поведенческие паттерны, поделены на следующие категории:

- цепочка обязанностей;
- команда;
- интерпретатор;
- итератор;
- посредник;
- хранитель;
- наблюдатель;
- состояние;
- стратегия;
- шаблонный метод;
- посетитель.

В качестве примера поведенческого паттерна рассмотрим «итератор». Это один из наиболее часто используемых паттернов, реализованный не в одном языке программирования.

Паттерн «итератор»

Итераторы предоставляют стандартный механизм последовательного обращения к коллекции, например к вектору. Функциональность может быть обеспечена так, чтобы к каждому элементу коллекции можно было получить доступ последовательно. Паттерн «итератор» предоставляет сокрытие данных, поддерживая таким образом безопасность внутренней структуры коллекции. Паттерн «итератор» также предусматривает возможность создания более одного итератора, каждый из которых будет функционировать без помех для остальных. В Java применяется собственная реализация итератора. Приведенный ниже код создает вектор, затем задает количество строк в нем:

```
package Iterator;

import java.util.*;
public class Iterator {
    public static void main(String args[]) {

        // Instantiate an ArrayList.
        ArrayList<String> names = new ArrayList();

        // Add values to the ArrayList
```

```
        names.add(new String("Joe")) ;
        names.add(new String("Mary"));
        names.add(new String("Bob")) ;
        names.add(new String("Sue")) ;

        //Now Iterate through the names
        System.out.printin("Names:") ;
        iterate(names) ;
    }

    private static void iterate(ArrayList<String> arl) {
        for(String listItem : arl) {
                    System.out.printin(listItem.toString());
                }
    }
}
```

Затем мы создаем такое перечисление, чтобы можно было выполнить его итерацию. Метод `iterate()` предназначен для выполнения итерации. В данном методе применяется метод перечисления Java `hasMoreElements()`, который обеспечивает последовательный доступ к вектору и выводит перечень всех названий.

Антипаттерны

Несмотря на то что паттерны проектирования развиваются из положительного опыта, существуют также антипаттерны, которые отражают неудачный опыт. Существует достаточно документальных свидетельств о том, что большинство проектов в области разработки программного обеспечения в конце концов завершается провалом. Как отмечено в статье «Создание хаоса» Джонни Джонсона («Creating Chaos», Johnny Johnson), добрая треть всех проектов полностью прекращается. Очевидно, что многие из этих провалов произошли из-за плохих проектных решений

Термин *антипаттерн* происходит из того, что паттерны проектирования создаются для решения определенного типа задач в рамках некоторого часто возникающего контекста. Антипаттерн же является реакцией на задачу и появляется благодаря неудачному опыту. В то время как паттерны проектирования полагаются на принципы SOLID и успешные случаи применения, антипаттерны можно считать конструкциями, которые нужно избегать.

В докладе по C++ в ноябре 1995 года Эндрю Кениг (Andrew Koenig) назвал два основных признака антипаттернов:

- Они предлагают неправильное решение задачи, которое ведет к неудаче.
- В то же время они предлагают выход из положения и способы перехода от неудачного решения к успешному.

Многие разработчики уверены, что антипаттерны полезнее самих паттернов проектирования. Так происходит из-за того, что антипаттерны разработаны для решения проблем, которые уже появились. Это сводится к концепции анализа первопричины. Можно провести исследования с данными, которые могут указывать, почему исходное исполнение, возможно, являющееся паттерном проектирования, оказалось неудачным. Можно сказать, что антипаттерны возникают из провала предыдущих решений. Поэтому антипаттерны позволяют рассмотреть проблему в ретроспективе.

Например, в статье «Повторное использование паттернов и антипаттернов» Скот Эмблер описал паттерн, названный «надежный артефакт», и дал ему такое определение:

> Конструкция с хорошей документацией, созданная для удовлетворения общих потребностей, а не для конкретных потребностей проекта, тщательно протестированная и снабженная несколькими примерами, чтобы показать, как с ней работать. Код с такими качествами гораздо более приспособлен к повторному использованию, чем код без таких качеств. «Надежный артефакт» — это конструкция, которую легко понять и с которой легко работать.

Однако, конечно, происходит много случаев, когда заявлено, что решение пригодно для повторного использования, но никто никогда не использует его повторно. Таким образом, чтобы дать понимание антипаттерна, он пишет:

> Кто-то помимо изначального разработчика должен пересмотреть «непригодный артефакт», чтобы определить, может ли кто-нибудь проявить к нему интерес. Если да, то такой артефакт нужно переделать в «надежный».

Таким образом, антипаттерны позволяют пересматривать существующие конструкции проектирования и проводить непрерывную реорганизацию кода до тех пор, пока не будет найдено работающее решение.

НЕКОТОРЫЕ УМЕСТНЫЕ ПРИМЕРЫ АНТИПАТТЕРНОВ

- Одиночка.
- Локатор служб.
- Магические числа/строки.
- Раздувание интерфейса.
- Кодирование путем исключения.
- Сокрытие/проглатывание ошибок.

Заключение

В этой главе мы рассмотрели концепцию паттернов проектирования. Паттерны — это часть повседневной жизни разработчика, а также способ мышления в объектно-ориентированной разработке. Как и многое, что относится к информационным технологиям, корни решений создаются в окружающей действительности.

Хотя в этой главе дано только краткое описание паттернов проектирования, настоятельно советуем углубиться в эту тему, ознакомившись с какой-нибудь из соответствующих книг.

Ссылки

Александер Кристофер с соавт. «Язык шаблонов. Города. Здания. Строительство» (A Pattern Language: Towns, Buildings, Construction). Кембридж, Великобритания: Издательство Оксфордского Университета, 1977.

Эмблер Скотт. «Повторное использование паттернов и антипаттернов» (Reuse Patterns and Antipatterns) // Журнал разработчика программного обеспечения, 2000.

Гамма Э., Хелм Р., Джонсон Р., Влиссидес Дж. Приемы объектно-ориентированного проектирования. Паттерны проектирования. — СПб.: Питер, 2020. — 368 с.: ил. (Серия «Библиотека программиста»).

Грэнд Марк. «Паттерны в Java: Каталог повторно используемых паттернов проектирования с пояснениями в UML-схемах» (Patterns in Java: A Catalog of Reusable Design Patterns Illustrated with UML). — 2-е изд., том 1. Хобокен, Нью-Джерси: Вайли, 2002.

Яворски Джейми. «Реализация платформы Java 2» (Java 2 Platform Unleashed). Индианаполис, Индиана: Sams Publishing, 1999.

Джонсон Джонни. «Создание Хаоса» (Creating Chaos) // Американский программист, 1995 год, июль.

Ларман, Крэг. «Применение UML и Паттернов: Введение в объектно-ориентированный анализ, проектирование и итеративная разработка» (Applying UML and Patterns: An Introduction to Object-Oriented Analysis and Design and Iterative Development), 3-е изд. Хобокен, Нью-Джерси: Вайли, 2004.

Глава 11

ИЗБЕГАНИЕ ЗАВИСИМОСТЕЙ И ТЕСНО СВЯЗАННЫХ КЛАССОВ

Как уже говорилось в главе 1 «Введение в объектно-ориентированные концепции», традиционные критерии классического объектно-ориентированного программирования — это инкапсуляция, наследование и полиморфизм. В теории, чтобы рассматривать язык программирования как объектно-ориентированный, он должен соответствовать этим трем принципам. Кроме того, как уже отмечалось в главе 1, лично я предпочитаю дополнить список композицией.

Таким образом, когда я провожу обучение объектно-ориентированному программированию, мой список фундаментальных концепций выглядит так:

- Инкапсуляция.
- Наследование.
- Полиморфизм.
- Композиция.

ПОДСКАЗКА

Возможно, в этом списке не хватает интерфейсов, но я уже принимаю интерфейсы за особый тип наследования.

Дополнение этого списка композицией даже более важно в среде разработчиков на сегодняшний день, поскольку не прекращается спор о том, как правильно использовать наследование. Проблема использования интерфейсов достаточно давняя. В последние несколько лет споры только усилились.

Многие разработчики, с которыми я беседовал, ратуют за использование композиции вместо наследования (что часто называют «композиция против наследования»). На самом деле, некоторые разработчики полностью избегают наследования или, по крайней мере, ограничивают использование наследования единственным уровнем иерархии.

Причина столь большого внимания к использованию наследования происходит от проблемы связывания. В пользу использования наследования говорит, безусловно, возможность повторного использования, расширяемость и полиморфизм, в то же время наследование может привести к проблемам возникновения зависимостей между классами, а именно их связыванию. Такие зависимости сулят проблемы при сопровождении и тестировании.

В главе 7 «Наследование и композиция» обсуждалось, как наследование может на самом деле ослабить инкапсуляцию, что звучит нелогично, поскольку и то и другое является фундаментальной концепцией. Тем не менее это лишь часть всего веселья, которая заключается в том, что действительно нужно думать, как правильно применить наследование.

ПРЕДОСТЕРЕЖЕНИЕ

Имейте в виду, что я не сторонник избегания наследования. Речь прежде всего идет о том, что нужно избегать зависимостей и тесно связанных классов. Когда именно имеет смысл использовать наследование — важный предмет дискуссий.

Такие разговоры ведут к тому, что встает вопрос: если не наследование, то что тогда? Можно использовать композицию. Вряд ли это кого-то удивит, потому что по ходу книги я не прекращаю утверждать, что в реальности есть только два способа повторного использования классов: использование наследования и использование композиции. Можно создать дочерний класс с помощью наследования от родительского либо можно включить один класс в состав другого посредством композиции.

Если, как утверждают многие, наследования необходимо избегать, зачем мы тратим на него время, изучая? Ответ прост: много кода написано с использованием наследования. Как вскоре понимает большинство разработчиков, преобладающее количество кода встречается именно при сопровождении. Таким образом, необходимо понимание того, как исправить, улучшить и правильно сопровождать код, написанный с использованием наследования. Может даже быть такое, что вы сами напишете новый код с наследованием. Проще говоря, программисту нужно охватить все возможные основы и полностью изучить инструментарий разработчика. Однако это также означает, что придется расширять набор используемых инструментов, а еще то, что нужно переосмыслить, как мы их будем использовать.

И повторюсь, поймите, пожалуйста, что я не хотел выносить никаких оценочных суждений. Я не могу утверждать, что от наследования одни проблемы и лучше совсем без него. Я хочу сказать о том, что важно в полной мере понять, как используется наследование, вдумчиво изучить альтернативные способы исполнения, а затем уже самим для себя решить. Следовательно, цель примеров в этой главе не в том, чтобы предложить оптимальный способ проектирования классов. Это лишь примеры с целью дать представление о проблемах, связанных с вы-

бором между наследованием и композицией. Стоит помнить, что для всех технологий важно развитие: сохранение хорошего и отсеивание того, что не очень.

Более того, у композиции есть свои проблемы связывания классов. В главе 7 я говорил о различных типах композиций: ассоциации и агрегации. Агрегации представляют собой объекты, встроенные в другие объекты (созданные с помощью ключевого слова `new`), в то время как ассоциации являются объектами, которые попадают внутрь других объектов через список параметров. Поскольку агрегации встроены в объекты, они тесно связаны, а этого мы хотим избежать.

Поэтому в то время как наследование получило в глазах многих репутацию причины тесного связывания классов, композиция (при использовании агрегаций) также может привести к тесному связыванию. Вернемся к примеру с компонентами стереосистемы, который приводился в главе 9 «Создание объектов и объектно-ориентированное проектирование» затем, чтобы свести воедино эти концепции в конкретный пример.

Создание стереосистемы с помощью агрегации приведет к созданию магнитолы, компоненты которой интегрированы в единую систему. Это может быть удобно во многих ситуациях. Ее можно взять с собой, легко переносить, а еще не надо ничего подключать. Однако такое исполнение может принести много проблем. Если один компонент, скажем MP3-плеер, выйдет из строя, то в ремонт придется нести все устройство целиком. Что еще хуже, вся система может разом выйти из строя и стать непригодной к дальнейшему использованию, например, из-за скачка напряжения.

Создание стереосистемы методом ассоциации может помочь снизить риск проблем, которые возникнут при агрегации. Можно рассмотреть стереосистему, состоящую из многих компонентов, как связку ассоциаций, объединенных проводами или беспроводной связью. При таком исполнении присутствует центральный объект, называемый приемником, который подключен к некоторым другим объектам, например динамикам, CD-плеерам, даже, допустим, к магнитофонам или граммофонам. На самом деле это можно рассмотреть как решение, не зависящее от выбора поставщика, поскольку у нас есть основное преимущество — использовать тот компонент, который нас устраивает.

В таком случае, если выйдет из строя CD-плеер, его можно будет просто отключить, при этом будет оставаться возможность либо его починить (используя систему временно без него), либо просто заменить на новый рабочий. В этом заключается преимущество ассоциаций — сведения связывания классов к минимуму.

ПОДСКАЗКА

Как уже было замечено в главе 9, хотя, как правило, тесно связанные классы не одобряются, бывают случаи, когда имеет смысл смириться с риском тесного связывания. Магнитола — это лишь один из примеров. Несмотря на то что у нее тесно связанное исполнение, она иногда может быть предпочтительнее.

Теперь, когда мы рассмотрели проблемы тесного связывания классов как при наследовании, так и при композиции, рассмотрим примеры тесно связанных конструкций с применением обоих методов. Подобно тому, как я веду лекции, мы будем неоднократно повторять эти примеры, пока не дойдем до техники, называемой «внедрение зависимостей», для смягчения последствий проблем связывания.

Композиция против наследования и внедрения зависимостей

Для начала можно сосредоточиться на том, как взять модель (из примеров, которые часто приводились в этой книге), построенную с помощью наследования, и перепроектировать ее уже с помощью композиции. Второй пример показывает, как можно перепроектировать класс с помощью композиции, пусть даже используя агрегацию, хотя не всегда это оптимальное решение. Третий пример показывает, как избегать агрегаций и применять ассоциации вместо них, и дает представление о концепции *внедрения зависимостей*.

1. Наследование

Независимо от того, интересует ли вас спор о том, чему отдавать свое предпочтение, начнем с простого примера наследования и рассмотрим способы, как можно было выполнить конструкцию с применением композиции. Для этого вернемся к примеру с классом `Mammal`, который неоднократно уже встречался на страницах этой книги.

В этом случае введем класс `Bat` (летучая мышь) — млекопитающее, но умеющее летать, как можно увидеть на рис. 11.1.

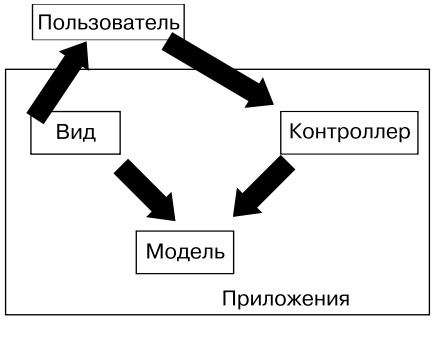

Рис. 11.1. Использование наследования для создания классов млекопитающих

В частности, в этом примере выбор наследования является очевидным. Класс Dog, который мы создаем, наследует от класса Mammal, верно? Посмотрите на код ниже, который использует наследование вот так:

```
class Mammal {
    public void eat () {System.out.printin("I am Eating") ;};
}
class Bat extends Mammal {
    public void fly () {System.out.printin("I am Flying") ;};
}
class Dog extends Mammal {
    public void walk () {System.out.printIn("I am Walking") ;};
}
public class TestMammal {

    public static void main(String args[]) {

        System.out.printIn("Composition over Inheritance") ; ;

        System.out.print1n("\nDog") ;
        Dog fido = new Dog();
        fido.eat();
        fido.walk();

        System.out.printIn("\nBat") ;
        Bat brown = new Bat();
        brown.eat();
        brown. fly();

    }

}
```

В этой конструкции класс Mammal имеет одно поведение, eat(), предполагая, что все млекопитающие принимают пищу. Однако мы заметим проблему наследования сразу же, как добавим два подкласса Mammal — Bat и Dog. Собака-то может ходить, а вот могут ли все другие млекопитающие? К тому же летучая мышь может передвигаться в воздухе, хотя далеко не все остальные млекопитающие могут. Вопрос состоит в том, как работают эти методы. По аналогии с приведенным ранее примером с пингвином (напомним, что не все птицы умеют летать), принятие решения о местоположении того или иного метода в иерархии наследования подчас вызывает затруднение.

Разделение класса Mammal на FlyingMammals и WalkingMammals — не особо изящное решение, поскольку это только верхушка пресловутого айсберга. Есть млекопитающие, которые плавают, есть даже те, которые откладывают яйца. Кроме того, явно можно понять, что существует бесчисленное множество других черт поведения, присущих тем или иным видам. Непрактично создавать отдельный

класс для каждого возможного поведения. Получается, вместо отношения «является экземпляром» в такой конструкции, пожалуй, лучше применить отношение «содержит как часть».

2. Композиция

При такой стратегии вместо непосредственного встраивания поведений в классы мы создаем отдельные классы для каждого поведения. Потому, вместо того чтобы помещать поведения в иерархию наследования, можно создать классы для каждого поведения и уже потом создавать модели разных млекопитающих, задействовав только те поведения, которые необходимы (с помощью агрегации).

Таким образом, мы создаем класс `Walkable` и класс `Flyable`, как показано на рис. 11.2.

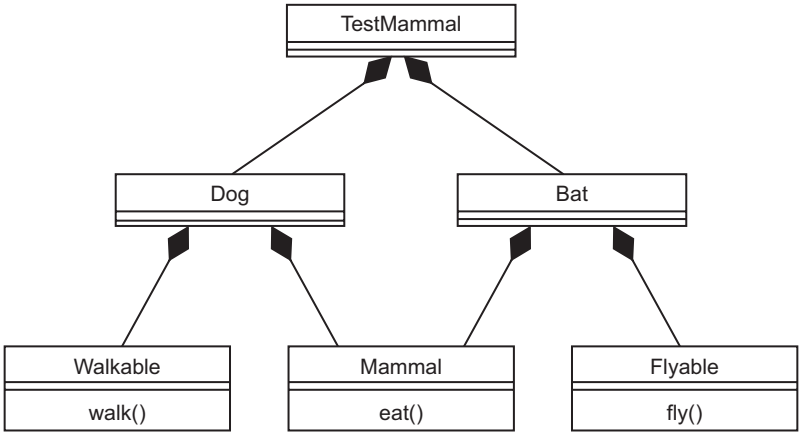

Рис. 11.2. Использование композиции для создания классов млекопитающих.

Например, взгляните на следующий код. У нас по-прежнему есть класс `Mammal` с методом `eat()`, а еще есть классы `Dog` и `Bat`. Главное различие при проектировании в том, что поведения классов `Dog` и `Bat` задаются с помощью композиции, а именно агрегации.

ПРЕДОСТЕРЕЖЕНИЕ

Обратите внимание на то, что мы недавно уже упоминали агрегацию. Этот пример объясняет нам, как композиция используется вместо наследования. Опять же, в этом примере применяется агрегация со значительной степенью связывания классов. Соответственно, рассмотрим его как промежуточный общеразвивающий этап, ведущий к следующему примеру, в котором будут применяться интерфейсы.

```
class Mammal {
    public void eat () {System.out.printin("I am Eating") ;};
}
class Walkable {
    public void walk () {System.out.printin("I am Walking") ;};
}
class Flyable {
    public void fly () {System.out.printIn("I am Flying") ;};
}
class Dog {
    Mammal dog = new Mammal () ;
    Walkable walker = new Walkable();
}
class Bat {
    Mammal bat = new Mammal () ;
    Flyable flyer = new Flyable();
}

public class TestMammal {

    public static void main(String args[]) {

        System.out.printIn("Composition over Inheritance") ;;
        System.out.printin("\nDog") ; ;
        Dog fido = new Dog();
        fido.dog.eat();
        fido.walker.walk();

        System.out.printin("\nBat") ; ;
        Bat brown = new Bat();
        brown.bat.eat();
        brown.flyer.fly();
    }
}
```

ПРИМЕЧАНИЕ

Целью этого примера является наглядно объяснить, как применять композицию вместо наследования. Но это совсем не значит, что нигде в проектировании не нужно применять наследование. Если вы полагаете, что все млекопитающие едят, то, например, можно решить поместить метод `eat ()` в класс `Mammal` и передать этот метод с помощью наследования классам `Dog` и `Bat`. Как правило, это решение зависит от выбранного исполнения.

Есть вероятность, что в основе споров лежит концепция, о которой говорилось ранее, что наследование нарушает инкапсуляцию. Это нетрудно понять, потому что изменение в классе `Mammal` будет требовать перекомпиляции (возможно, даже повторного внедрения) всех подклассов `Mammal`. Это означает, что классы связаны тесно, а это противоречит поставленной цели — обеспечить как можно меньшее связывание классов.

В нашем примере с композицией, если бы мы захотели добавить класс `Whale` (кит), не пришлось бы переписывать никакие из уже созданных классов. Тогда бы мы просто добавили класс `Swimmab1e` и класс `Whale`.

Класс `Swimmab1e` можно повторно использовать, скажем, для класса `Dolphin` (дельфин).

```
class Swimmable {
    public void fly () {System.out.printiIn("I am Swimming") ;};
}
class Whale {
    Mammal whale = new Mammal();
    Walkable swimmer = new Swimmable ();
}
```

Добавление функционала с помощью основного приложения может произойти без изменений уже существующих классов.

```
System.out.printIn("\nWhale");
Whale shamu = new Whale();
shamu.whale.eat();
shamu.swimmer.swim();
```

Одно из железных правил — использовать наследование только в случаях, когда по-настоящему присутствует полиморфизм. Например, при создании классов `Circle` и `Rectangle`, наследующих от класса `Shape`, вполне разумно использовать наследование. С другой стороны, поведения, такие как умение ходить и летать, вряд ли хорошо пригодны для применения наследования, поскольку их переопределение может принести проблемы. Например, если нужно скорректировать метод `f1y()` для класса `Dog`, единственным очевидным способом будет команда (no-op) — не выполнять никаких действий. Опять же, как и в примере с классом `Penguin`, нам не нужно, чтобы класс `Dog` мог летать по воздуху и выполнялся присутствующий метод `fly()`. Поэтому, к большому разочарованию нашего Тузика, метод `fly()` не приведет ни к каким реальным действиям.

Хотя в этом примере действительно применена композиция, такое исполнение имеет важный изъян. Объекты тесно связаны, поскольку мы видим использование ключевого слова `new`.

```
class Whale {
    Mammal whale = new Mammal () ;
    Walkable swimmer = new Swimmable ();
}
```

В завершение нашего упражнение по снижению связанности классов введем концепцию внедрения зависимостей. Простым языком, вместо того чтобы создавать объекты внутри других, мы будем внедрять объекты извне с помощью списка параметров. Разговор будет строиться только вокруг концепции внедрения зависимостей.

Внедрение зависимостей

Пример в предыдущем разделе использует композицию (с агрегацией) для того, чтобы наделить класс `Dog` способностью ходить — поведением `Walkable`. Класс `Dog` буквально порождает новый объект `Walkab1e` внутри самого класса `Dog`, как показано в следующем фрагменте кода:

```
class Dog {
    Walkable walker = new Walkable();
}
```

Хотя это вполне рабочий вариант, классы остаются тесно связанными. Чтобы полностью разделить классы в предыдущем примере, нужно привести в действие концепцию внедрения зависимостей, о которой уже упоминалось. Внедрение зависимостей и инверсия управления часто рассматриваются вместе. Одно из назначений инверсии управления (IOC) — назначить кого-либо ответственным за создание экземпляра зависимости и передачу его впоследствии вам. Это именно то, что мы собираемся реализовать в данном примере.

Поскольку не все млекопитающие могут ходить, летать или плавать, требуется создать интерфейсы, воссоздающие поведения разного рода млекопитающих, чтобы начать разделение связанных классов. В этом примере я сосредоточусь на поведении, отвечающем за умение ходить, и создам интерфейс `IWalkab1e`, как показано на рис. 11.3.

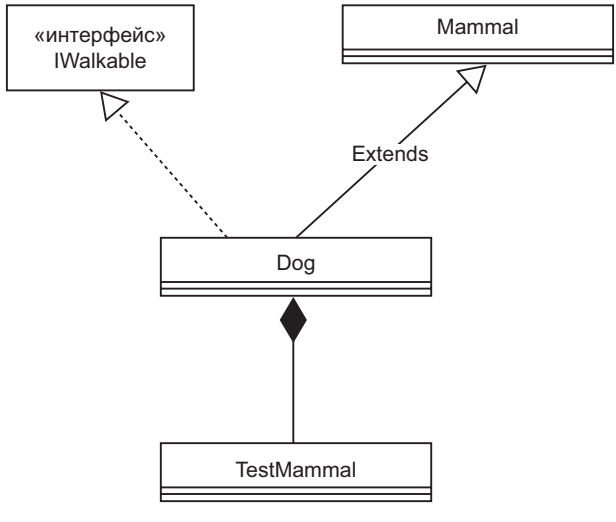

Рис. 11.3. Использование интерфейсов для создания классов млекопитающих

Код интерфейса `IWalkable` выглядит так:

```
interface IWalkable {
    public void walk();
}
```

Единственный метод этого интерфейса — это `walk()`, который оставили определенному классу для обеспечения реализации.

```
class Dog extends Mammal implements IWalkable;
    Walkable walker;
    public void setWalker (Walkable w) {
        this.walker=w;
    }
    public void walk () {System.out.printIn("I am Walking");};
}
```

Обратите внимание, что класс `Dog` является расширением класса `Mammal` и обеспечивает реализацию интерфейса `IWalkable`. Также обратите внимание, что класс `Dog` предоставляет ссылку, а конструктор — механизм для внедрения зависимости.

```
Walkable walker;
public void setWalker (Walkable w) {
    this.walker=w;
}
```

Вот что такое внедрение зависимостей в общих чертах. Поведение `Walkable` не создается внутри класса `Dog` с помощью ключевого слова `new`, а внедряется в этот класс через список параметров.

Приведем полный пример:

```
class Mammal {
    public void eat () {System.out.printin("I am Eating") ;};
}
interface IWalkable {
    public void walk();
}
class Dog extends Mammal implements IWalkable{
    Walkable walker;
    public void setWalker (Walkable w) {
        this.walker=w;
    }
    public void walk () {System.out.printin("I am Walking") ;};
}

    public class TestMammal {
```

```
        public static void main(String args[]) {

        System.out.printIn("Composition over Inheritance") ;
        System.out.printin("\nDog") ;

        Walkable walker = new Walkable();
                Dog fido = new Dog();
        fido.setWalker (walker) ;
        fido.eat();
        fido.walker.walk();

    }

}
```

Несмотря на то что в примере используется внедрение с помощью конструктора, это не единственный способ провести внедрение зависимостей.

Внедрение с помощью конструктора

Один из способов внедрения поведения `Walkable` — создание конструктора внутри класса `Dog`, который при вызове будет принимать аргумент от основного приложения следующим образом:

```
class Dog {
    Walkable walker;
    public Dog (Walkable w) {
    this.walker=w;
    }
}
```

При таком подходе приложение создает экземпляр объекта `Walkable` и вставляет его в класс `Dog` с помощью конструктора.

```
Walkable walker = new Walkable();

Dog fido = new Dog(walker)
```

Внедрение с помощью сеттера

Хотя конструктор будет инициализировать атрибуты во время создания объекта, на протяжении всего существования объекта зачастую приходится перезагружать значения. Здесь применяются методы мутаторов — в форме сеттеров. Поведение `Walkable` можно вставить в класс `Dog` с помощью сеттера, в данном случае `setWalker()`:

```
class Dog {
    Walkable walker;
    public void setWalker (Walkable w) {
        this.walker=w;
    }
}
```

Благодаря конструктору приложение создает объект `Walkable` и вставляет его в класс `Dog` с помощью сеттера:

```
Walkable walker = new Walkable();
Dog fido = new Dog();
fido.setWalker (walker);
```

Заключение

Внедрение зависимостей снижает связанность конструкции класса благодаря избавлению от зависимостей. Это что-то вроде покупки готовой продукции (от поставщика) вместо создания экземпляра каждый раз собственноручно.

Этот вопрос играет ключевую роль в споре о выборе между наследованием и композицией. Важно заметить, что это лишь обсуждение. Цель этой главы заключается не столько в описании «оптимального» способа проектирования классов, сколько в настраивании на мышление в отношении проблем, связанных с выбором между наследованием и композицией. В следующей главе мы углубимся в изучение принципов объектно-ориентированного проектирования SOLID, набора концепций, который так высоко признается и ценится сообществом разработчиков программного обеспечения.

Ссылки

Мартин Роберт с соавт. «Гибкая разработка программного обеспечения: принципы, паттерны и инструкции» (Agile Software Development, Principles, Patterns, and Practices). Бостон, штат Массачусетс: PearsonEducation, Inc., 2002.

Мартин Р. Чистый код: создание, анализ и рефакторинг. Библиотека программиста. — СПб.: Питер, 2018. — 464 с.: ил.

Глава 12

ПРИНЦИПЫ ОБЪЕКТНО-ОРИЕНТИРОВАННОГО ПРОЕКТИРОВАНИЯ SOLID

Одно из наиболее распространенных утверждений, которые делают многие разработчики в отношении объектно-ориентированного программирования, заключается в том, что его основным преимуществом является моделирование реального мира. Я признаю, что часто использую эти слова, когда рассказываю о классических концепциях объектно-ориентированного программирования. Как считает Роберт Мартин (по меньшей мере так он утверждал в одной из своих лекций на YouTube), основная идея объектно-ориентированного проектирования близка к тому взгляду, который распространен в маркетинге. Между тем он утверждает, что объектно-ориентированная разработка в основном касается управления зависимостями посредством инверсии ключевых зависимостей в целях предотвращения негибкости кода, его недолговечности, а также невозможности повторного использования кода.

Например, в курсах по классическому объектно-ориентированному программированию код часто прямо повторяет ситуации, встречающиеся в жизни. Например, если собака — млекопитающее, то очевидно, что для этой связи лучшим выбором будет наследование. Точная проверка на отношение «содержит как часть» и «является экземпляром», подобная лакмусовой реакции, является частью объектно-ориентированного мышления долгие годы.

Однако, как мы видели уже на протяжении этой книги, попытки внедрить отношение наследования может вызвать проблемы проектирования (вспомните пример с собаками, которые не лают). Действительно ли при проектировании стоит отделить собак, не умеющих лаять, от тех, которые умеют, или летающих птиц от нелетающих при помощи наследования? Было ли это все создано объектно-ориентированными маркетологами? Хорошо, не нужно шумихи. Как мы видели в предыдущей главе, судя по всему, строгая однобокость в выборе между отношениями *содержит как часть* и *является экземпляром* далеко не

всегда является лучшим подходом. Похоже, нам стоит больше обращать внимание на разделение связанных классов.

В лекции, о которой я уже упоминал, Роберт Мартин, которого часто называют Дядей Бобом, использует следующие три термина для описания кода, непригодного для повторного использования:

- **Негибкость** — когда изменение в одной части программы может вызвать сбой в другой части.
- **Недолговечность** — когда что-либо «ломается» в местах, не связанных между собой.
- **Ограниченная подвижность** — когда код нельзя повторно использовать вне оригинального контекста.

Принципы SOLID были созданы для устранения подобных ограничений и достижения высокого качества кода. Роберт Мартин ввел эти пять принципов объектно-ориентированной разработки, чтобы «придать исполнению больше рациональности, гибкости и сопровождаемости». По словам Мартина, принципы SOLID также формируют ядро философии различных методик, например гибкой методологии объектно-ориентированной разработки или адаптивной разработки. Сокращение SOLID появилось благодаря Майклу Фезерсу.

Ниже перечислены пять принципов SOLID:

- **Single Responsibility Principle (SRP)** — принцип единственной ответственности.
- **Open-Closed Principle (OCP)** — принцип открытости/закрытости.
- **Liskov Substitution Principle (LSP)** — принцип подстановки Барбары Лисков.
- **Interface Segregation Principle (ISP)** — принцип разделения интерфейса.
- **Dependency Inversion Principle (DIP)** — принцип инверсии зависимостей.

В главе рассказывается об этих самых пяти принципах и показана их связь с принципами классического объектно-ориентированного программирования, которые играли важную роль на протяжении десятилетий. Я постараюсь объяснить принципы SOLID на максимально простых примерах. В глобальной сети есть много материала на эту тему, в том числе несколько неплохих видео на YouTube. Многие из этих видео нацелены на разработчиков и не всегда будут понятны студентам и новичкам.

Как и во всех приведенных ранее примерах в этой книге, я постараюсь не усложнять, а дать суть концепций, максимально упростив конструкцию в образовательных целях.

Принципы объектно-ориентированной разработки SOLID

В главе 11 «Избегание зависимостей и тесно связанных классов» мы обсуждали некоторые фундаментальные концепции, постепенно подбираясь к обсуждению пяти принципов SOLID. В этой главе мы подробно рассмотрим каждый принцип SOLID. Все характеристики принципов SOLID взяты с сайта Дяди Боба: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

1. SRP: принцип единственной ответственности

Принцип единственной ответственности гласит о том, для внесения изменений в класс требуется только одна причина. Каждый класс и модуль программы должны иметь в приоритете одно задание. Поэтому не стоит вносить методы, которые могут вызвать изменения в классе более чем по одной причине. Если описание класса содержит слово «and», то принцип SRP может быть нарушен. Другими словами, каждый модуль или класс должен нести ответственность за одну какую-либо часть функционала программного обеспечения, и такая ответственность должна быть полностью инкапсулирована в класс.

Создание иерархии фигур — это один из классических примеров, иллюстрирующих наследование Этот пример часто встречается в обучении, а я использую его на протяжении этой главы (равно как и всей книги). В этом примере класс `Circle` наследует атрибуты от класса `Shape`. Класс `Shape` предоставляет абстрактный метод `calcArea()` в качестве контракта для подкласса. Каждый класс, наследующий от `Shape`, должен иметь собственную реализацию метода `calcArea()`:

```
abstract class Shape{
    protected String name;
    protected double area;
    public abstract double calcArea();
}
```

В этом примере класс `Circle`, наследующий от класса `Shape`, при необходимости обеспечивает свою реализацию метода `calcArea()`:

```
class Circle extends Shape{
    private double radius;

    public Circle(double r) {
        radius = r;
    }
    public double calcArea() {
        area = 3.14*(radius*radius) ;
        return (area);
    };
}
```

ПРЕДОСТЕРЕЖЕНИЕ

В этом примере мы только собираемся рассмотреть класс `Circle`, чтобы сосредоточиться на принципе единственной ответственности и сделать пример максимально простым.

Третий класс, `CalculateAreas`, подсчитывает площади различных фигур, содержащихся в массиве `Shape`. Массив `Shape` обладает неограниченным размером и может содержать различные фигуры, например квадраты и треугольники.

```
class CalculateAreas {
    Shape[] shapes;
    double sumTotal=0;
    public CalculateAreas(Shape[] sh) {
        this.shapes = sh;
    }
    public double sumAreas() {
        sumTotal=0;
        for (inti=0; i<shapes.length; i++) {
        sumTotal = sumTotal + shapes[i].calcArea() ;
        }
        return sumTotal ;
    }
    public void output() {
        System.out.printIn("Total of all areas = " + sumTotal);
    }
}
```

Обратите внимание, что класс `CalculateAreas` также обрабатывает вывод приложения, что может вызвать проблемы. Поведение подсчета площади и поведение вывода связаны, поскольку содержатся в одном и том же классе.

Мы можем проверить работоспособность этого кода с помощью соответствующего тестового приложения `TestShape`:

```
public class TestShape {
    public static void main(String args[]) {

        System.out.printin("Hello World!");

        Circle circle = new Circle(1);

        Shape[] shapeArray = new Shape[1];
        shapeArray[0] = circle;

        CalculateAreas ca = new CalculateAreas(shapeArray) ;

        ca.sumAreas() ;
        ca.output();
    }
}
```

Теперь, имея в распоряжении тестовое приложение, мы можем сосредоточиться на проблеме принципа единственной ответственности. Опять же, проблема связана с классом `CalculateAreas` и с тем, что этот класс содержит поведения и для *сложения* площадей различных фигур, а также для *вывода* данных.

Основополагающий вопрос (и, собственно, проблема) в том, что если нужно изменить функциональность метода `output()`, потребуется внести изменения в класс `CalculateAreas` независимо от того, изменится ли метод подсчета площади фигур. Например, если мы вдруг захотим осуществить вывод данных в HTML-консоль, а не в простой текст, нам потребуется заново компилировать и повторно внедрять код, который складывает площади фигур. Все потому, что ответственности связаны.

В соответствии с принципом единственной ответственности, задача состоит в том, чтобы изменение одного метода не повлияло на остальные методы и не приходилось проводить повторную компиляцию. «У класса должна быть одна, только одна, причина для изменения — единственная ответственность, которую нужно изменить».

Чтобы решить данный вопрос, можно поместить два метода в отдельные классы, один для оригинального консольного вывода, другой для вывода в HTML:

```
class CaiculateAreas {;
    Shape[] shapes;
    double sumTotal=0;

    public CalculateAreas(Shape[] sh) {
        this.shapes = sh;
    }

    public double sumAreas() {
        sumTotal=0;

        for (inti=0; i<shapes.length; i++) {

            sumTotal = sumTotal + shapes[i].calcArea();

        }

            return sumTotal;
        }
}
class OutputAreas {
    double areas=0;
    public OutputAreas (double a) {
        this.areas = a;
    }

    public void console() {
```

```
            System.out.printin("Total of all areas = " + areas);
        }
        public void HTML() {
            System.out.printIn("<HTML>") ;
            System.out.printin("Total of all areas = " + areas);
            System.out.printin("</HTML>") ;
        }
    }
```

Теперь с помощью недавно написанного класса мы можем добавить функциональность для вывода в HTML без воздействия на код для вычисления площади:

```
    public class TestShape {
        public static void main(String args[]) {

            System.out.printin("Hello World!");

            Circle circle = new Circle(1);

            Shape[] shapeArray = new Shape[1];
            shapeArray[0] = circle;

            CalculateAreas ca = new CalculateAreas(shapeArray) ;

            CalculateAreas sum = new CalculateAreas(shapeArray) ;
            OutputAreasoAreas = new OutputAreas(sum.sumAreas() ) ;

            oAreas.console(); // output to console
            oAreas.HTML() ; // output to HTML
        }
    }
```

Суть здесь заключается в том, что теперь можно послать вывод в различных направлениях в зависимости от необходимости. Если нужно добавить возможность другого способа вывода, например JSON, можно привнести ее в класс `OutputAreas` без необходимости внесения изменений в класс `CalculateAreas`. В результате можно перераспределить класс `CalculateAreas` без какого-либо затрагивания других классов.

2. OCP: принцип открытости/закрытости

Принцип открытости/закрытости гласит, что можно расширить поведение класса без внесения изменений.

Обратим снова внимание на пример с фигурами. В приведенном ниже коде есть класс `ShapeCalculator`, который берет объект `Rectangle`, рассчитывает площадь этого объекта и возвращает значения. Это простое приложение, но оно работает только с прямоугольниками.

```
class Rectangle{
    protected double length;
    protected double width;

    public Rectangle(double 1, double w) {
        length = 1;
        width = w;
    };
}
class CalculateAreas {

    private double area;

    public double calcArea(Rectangle r) {

        area = r.length * r.width;

        return area;

    }
}
public class OpenClosed {
    public static void main(String args[]) {

        System.out.printin("Hello World");

        Rectangle r = new Rectangle(1,2);

        CalculateAreas ca = new CalculateAreas ();

        System.out.printin("Area = "+ ca.calcArea(r));

    }
}
```

То, что это приложение работает только в случае с прямоугольниками, приводит к ограничению, которое наглядно объясняет принцип открытости/закрытости: если мы хотим добавить класс `Circle` к классу `CalculateArea` (изменить то, что он выполняет), нам нужно внести изменения в сам модуль. Очевидно, что это вступает в противоречие с принципом открытости/закрытости, который гласит, что мы не должны вносить изменения в модуль для изменения того, что он выполняет.

Чтобы соответствовать принципу открытости/закрытости, можно вернуться к уже проверенному примеру с фигурами, где создается абстрактный класс `Shape` а непосредственно фигуры наследуют от класса `Shape`, у которого есть абстрактный метод `getArea()`.

На данный момент можно добавлять столь много разных классов, сколько требуется, без необходимости внесения изменений непосредственно в класс `Shape` (например, класс `Circle`). Сейчас можно сказать, что класс `Shape` закрыт.

Код ниже обеспечивает реализацию решения для прямоугольников и кругов и позволяет создавать неограниченное количество фигур:

```
abstract class Shape {
    public abstract double getArea() ;
}
class Rectangle extends Shape
{
    protected double length;
    protected double width;

    public Rectangle(double 1, double w) {
        length = 1;
        width = w;
    };
    public double getArea() {
        return length*width;
    }

}
class Circle extends Shape
{
    protected double radius;

    public Circle(double r) {
        radius = r;
    };
    public double getArea() {
        return radius*radius*3.14;
    }
}
class CalculateAreas {
    private double area;

    public double calcArea(Shape s) {
        area = s.getArea();
        return area;
    }
}

public class OpenClosed {
    public static void main(String args[]) {

        System.out.printiIn("Hello World") ;

        CalculateAreas ca = new CalculateAreas() ;

        Rectangle r = new Rectangle(1,2);

        System.out.printIn("Area = " + ca.calcArea(r));

        Circle c = new Circle(3);
```

```
            System.out.printIn("Area = " + ca.calcArea(c));

        }
    }
```

Стоит заметить, что при такой реализации в метод `CalculateAreas()` не должны вноситься изменения при создании нового экземпляра класса `Shape`.

Можно масштабировать код, не переживая о существовании предыдущего кода. Принцип открытости/закрытости заключается в том, что следует расширять код с помощью подклассов так, чтобы изначальный класс не требовал правок. Однако само понятие «расширение» выступает противоречивым в некоторых обсуждениях, касающихся принципов SOLID. Развернуто говоря, если мы отдаем предпочтение композиции, а не наследованию, как это влияет на принцип открытости/закрытости?

При соответствии одному из принципов SOLID код может удовлетворять критериям других принципов SOLID. Например, при проектировании в соответствии с принципом открытости/закрытости код может подходить требованиям принципа единственной ответственности.

3. LSP: принцип подстановки Лисков

Согласно принципу подстановки Лисков, проектирование должно предусматривать возможность замены любого экземпляра родительского класса экземпляром одного из дочерних классов. Если родительский класс может выполнять какую-либо задачу, дочерний класс тоже должен мочь.

Рассмотрим некоторый код, который на первый взгляд корректен, тем не менее нарушает принцип подстановки Лисков. В коде, приведенном ниже, присутствует типовой абстрактный класс `Shape`. Класс `Rectangle`, в свою очередь, наследует атрибуты от класса `Shape` и переопределяет его абстрактный метод `calcArea()`. Класс `Square`, в свою очередь, наследует от `Rectangle`.

```
abstract class Shape{
    protected double area;

    public abstract double calcArea();
}
class Rectangle extends Shape{
    private double length;
    private double width;

    public Rectangle(double 1, double w) {
        length = 1;
        width = w;
    }
    public double calcArea() {
```

```
        area = length*width;
        return (area) ;
    };

}
class Square extends Rectangle{
    public Square(double s) {
        super(s, Ss);
    }
}

public class LiskovSubstitution {
    public static void main(String args[]) {

        System.out.printIn("Hello World") ;

        Rectangle r = new Rectangle(1,2);

        System.out.printin("Area = " + r.calcArea());

        Square s = new Square(2) ;

        System.out.printin("Area = " + s.calcArea());

    }
}
```

Пока что все хорошо: прямоугольник является экземпляром фигуры, поэтому ничего не вызывает беспокойства, поскольку квадрат является экземпляром прямоугольника, — и снова все правильно, правда?

Теперь зададим философский вопрос: а квадрат — это все-таки прямоугольник? Многие ответят утвердительно. Хотя и можно допустить, что квадрат — это частный случай прямоугольника, но его свойства будут отличаться. Прямоугольник является параллелограммом (противоположные стороны одинаковы), как и квадрат. В то же время квадрат еще и является ромбом (все стороны одинаковы), в то время как прямоугольник — нет. Поэтому различия есть.

Когда дело доходит до объектно-ориентированного проектирования, проблема не в геометрии. Проблема состоит в том, как именно мы создаем прямоугольники и квадраты. Вот конструктор для класса `Rectangle`:

```
public Rectangle(double 1, double w) {
    length = 1;
    width = w;
}
```

Очевидно, конструктор требует два параметра. Однако конструктору для класса `Square` требуется только один, несмотря даже на то, что родительский класс, `Rectangle`, требует два.

```
class Square extends Rectangle{
    public Square(double s) {
    super(s, Ss);
    }
```

В действительности функционал для вычисления площади немного различен в случае каждого из этих двух классов. То есть класс `Square` как бы имитирует `Rectangle`, передавая конструктору один и тот же параметр дважды. Может казаться, что такой обходной прием вполне годится, но на самом деле он может ввести в заблуждение разработчиков, сопровождающих код, что вполне чревато подводными камнями при сопровождении в дальнейшем. По меньшей мере это неувязка и, наверное, сомнительное дизайнерское решение. Когда один конструктор вызывает другой, неплохо взять паузу и пересмотреть конструкцию — возможно, дочерний класс построен ненадлежащим образом.

Как же найти выход из этой ситуации? Попросту говоря, нельзя осуществить подстановку класса `Square` вместо `Rectangle`. Таким образом, `Square` не должен быть дочерним классом `Rectangle`. Они должны быть отдельными классами.

```
abstract class Shape {
    protected double area;

    public abstract double calcArea();
}

class Rectangle extends Shape {

    private double length;
    private double width;

    public Rectangle(double 1, double w) {
        length = 1;
        width = w;
    }

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

class Square extends Shape {
    private double side;

    public Square(double s) {
        side = s;
    }
    public double calcArea() {
        area = side*side;
        return (area);
    };
```

```
}

public class LiskovSubstitution {
    public static void main(String args[]) {

        System.out.printIn("Hello World") ;

        Rectangle r = new Rectangle(1,2);

        System.out.printIn("Area = " + r.calcArea());

        Square s = new Square(2) ;

        System.out.printIn("Area = " + s.calcArea());

    }
}
```

4. ISP: принцип разделения интерфейса

Принцип разделения интерфейсов гласит о том, что лучше создавать много небольших интерфейсов, чем несколько больших.

В этом примере мы создаем единственный интерфейс, который включает в себя несколько поведений для класса `Mammal`, а именно `eat()` и `makeNoise()`:

```
interface IMammal {
    public void eat();
    public void makeNoise() ;
}
class Dog implements IMammal {
    public void eat() {
        System.out.printIn("Dog is eating");
    }
    public void makeNoise() {
        System.out.printIn("Dog is making noise");
    }
}
public class MyClass {
    public static void main(String args[]) {

        System.out.printIn("Hello World");

        Dog fido = new Dog();
        fido.eat();
        fido.makeNoise()
    }
}
```

Вместо создания единственного интерфейса для класса `Mammal` нужно создать раздельные интерфейсы для всех поведений:

```
interface IEat {
    public void eat();
}
interface IMakeNoise {
    public void makeNoise() ;
}
class Dog implements IEat, IMakeNoise {
    public void eat() {
        System.out.printIn("Dog is eating");
    }
    public void makeNoise() {
        System.out.printIn("Dog is making noise");
    }
}
public class MyClass {
    public static void main(String args[]) {

        System.out.printIn("Hello World") ;

        Dog fido = new Dog();
        fido.eat();
        fido.makeNoise();
    }
}
```

Мы отделяем поведения от класса `Mammal`. Получается, что вместо создания единственного класса `Mammal` посредством наследования (точнее, интерфейсов) мы переходим к проектированию, основанному на композиции, подобно стратегии, которой придерживались в предыдущей главе.

В нескольких словах, с таким подходом мы можем создавать экземпляры класса `Mammal` с помощью композиции, а не быть вынужденными использовать поведения, которые заложены в единственный класс `Mammal`. Например, предположим, что открыто млекопитающее, которое не принимает пищу, а вместо этого поглощает питательные вещества через кожу. Если мы произведем наследование от класса `Mammal`, содержащего поведение `eat()`, для нового млекопитающего это поведение будет излишним. При этом если все поведения будут заложены в отдельные одиночные интерфейсы, получится построить класс каждого млекопитающего в точности так, как задумано.

5. DIP: принцип инверсии зависимостей

Принцип инверсии зависимостей предполагает, что код должен зависеть от абстрактных классов. Часто может казаться, что термины «инверсия зависимостей» и «внедрение зависимостей» взаимозаменяемы, однако это ключевые термины, которые нужно ясно понимать при обсуждении этого принципа. Сейчас постараемся их объяснить:

- **Инверсия зависимости** — принцип инвертирования зависимостей.
- **Внедрение зависимостей** — акт инвертирования зависимостей.
- **Внедрение конструктора** — осуществление внедрения зависимостей с помощью конструктора.
- **Внедрение параметра** — выполнение внедрения зависимостей через параметр метода, например сеттера.

Цель инверсии зависимостей в том, чтобы зависимость была с чем-то абстрактным, а не конкретным.

Хотя в какой-то момент, очевидно, придется создать что-то конкретное, мы постараемся создать конкретный объект (используя ключевое слово `new`) вверх по цепочке как можно дальше, как, например, в методе `main()`. Пожалуй, чтобы обдумать все это, лучше вернуться к главе 8 «Фреймворки и повторное использование: проектирование с применением интерфейсов и абстрактных классов», где обсуждается загрузка классов во время выполнения, а также к главе 9 «Создание объектов и объектно-ориентированное проектирование», где речь идет о снижении связанности и создании небольших классов с ограниченными ответственностями.

Одной из целей принципа инверсии зависимостей является выбор объектов во время выполнения, а не во время компиляции. (Можно изменить поведение программы во время выполнения). Можно даже писать новые классы без надобности перекомпиляции уже существующих (собственно, можно писать новые классы и внедрять их).

Много оснований для споров приведено в главе 11 «Избегание зависимостей и тесно связанных классов». Рекомендуем опираться на нее по мере рассмотрения принципа инверсии зависимостей.

Шаг 1: начальный пример

В этом примере мы в очередной раз вернемся к одному из классических примеров в объектно-ориентированном проектировании, сопровождавшему нас по всей книге, — классу `Mammal`, наряду с классами `Dog` и `Cat`, которые от него наследуют. Класс `Mammal` абстрактен и содержит лишь метод `makeNoise()`.

```
abstract class Mammal
{
    public abstract String makeNoise();
}
```

Подклассы, например `Cat`, используют наследование для заимствования поведения класса `Mammal`, `makeNoise()`:

```
class Cat extends Mammal
{
    public String makeNoise()
    {
        return "Meow";
    }
}
```

Затем основное приложение создает экземпляр объекта и вызывает метод `makeNoise()`:

```
Mammal cat = new Cat();;

System.out.printin("Cat says " + cat.makeNoise());
```

Полное приложение для первого шага представлено в следующем коде:

```
public class TestMammal {
    public static void main(String args[]) {

        System.out.print]n("Hello World\n") ;

        Mammal cat = new Cat();;
        Mammal dog = new Dog();

        System.out.printIn("Cat says " + cat.makeNoise());
        System.out.printIn("Dog says " + dog.makeNoise());

    }
}
abstract class Mammal
{
    public abstract String makeNoise();
}
class Cat extends Mammal
{
    public String makeNoise()
    {
        return "Meow" ;
    }
}
class Dog extends Mammal
    {
    public String makeNoise()
    {
    return "Bark";
    }
}
```

Шаг 2: разделение поведений

У кода, приведенного выше, есть один потенциально серьезный недостаток: он связывает классы млекопитающих и поведения (`MakingNoise`). В отделении по-

ведений млекопитающих от самих классов млекопитающих может заключаться значительное преимущество. Поэтому мы создаем класс `MakingNoise`, который могут использовать как млекопитающие, так и не млекопитающие.

При такой модели классы `Cat`, `Dog` и `Bird` могут расширить класс `MakeNoise` и создать свое «звуковое» поведение в зависимости от своих потребностей, например, как в следующем фрагменте кода класса `Cat`:

```
abstract class MakingNoise
{
        public abstract String makeNoise() ;
}

class CatNoise extends MakingNoise
{
    public String makeNoise()
    {
         return "Meow";
    }
}
```

При разделении поведения `MakingNoise` и класса `Cat` можно применить класс `CatNoise` вместо нагромождения кода в самом классе `Cat`, как показано в следующем фрагменте кода:

```
abstract class Mammal
{
    public abstract String makeNoise();
}
class Cat extends Mammal
{
    CatNoise behavior = new CatNoise();
    public String makeNoise()
    {
        return behavior.makeNoise() ;
    }
}
```

Далее приведено полное приложение для второго шага:

```
public class TestMammal {
    public static void main(String args[]) {

        System.out.printIn("Hello World\n") ;

        Mammal cat = new Cat();;
        Mammal dog = new Dog();

        System.out.printIn("Cat says " + cat.makeNoise());
        System.out.printIn("Dog says " + dog.makeNoise());

    }
```

```
}

abstract class MakingNoise
{
    public abstract String makeNoise() ;
}
class CatNoise extends MakingNoise
{
    public String makeNoise()
    {
        return "Meow";
    }
}

class DogNoise extends MakingNoise
    {
    public String makeNoise()
    {
        return "Bark";
    }
}
abstract class Mammal
{
    public abstract String makeNoise() ;
}
class Cat extends Mammal
{
    CatNoise behavior = new CatNoise();
    public String makeNoise()
    {
        return behavior.makeNoise() ;
    }
}
class Dog extends Mammal
{
    DogNoise behavior = new DogNoise();
    public String makeNoise()
    {
        return behavior.makeNoise() ;
    }
}
```

Проблема заключается в том, что хотя мы и провели отделение главной части кода, мы до сих пор не достигли нашей цели — инверсии зависимостей, поскольку класс `Cat` до сих пор создает экземпляр поведения издания звуков классом `Cat`.

```
CatNoise behavior = new CatNoise();
```

Класс `Cat` связан с низкоуровневым модулем `CatNoise`. Другими словами, нужно допускать связывание класса `Cat` не с классом `CatNoise`, а с абстрактным

классом для издания шума. На самом деле класс Cat вместо создания экземпляра поведения издания звуков должен принимать поведение через внедрение.

Шаг 3: внедрение зависимостей

На этом завершающем этапе мы полностью отбросим все, что связано с наследованием при проектировании, и изучим, как применять внедрение зависимостей посредством композиции. Не нужно использовать иерархии наследования, которые являются одной из главных причин, почему концепция композиции против наследования становится все более актуальной. Проще скомпоновать подтип, чем создавать его из иерархической модели.

К примеру, в изначальной реализации классов Cat и Dog в основном лежит один и тот же код; они просто возвращают разный шум. Получается так, что значительная часть кода избыточна. Таким образом, если бы требовалось создать много различных млекопитающих, требовалось бы написать много кода для издания звуков.

Может быть и так, что лучшим решением будет вынести код издания звуков из класса Mammal.

По большому счету, скачок при таком проектировании будет заключаться в том, что теперь можно не привязываться к определенным млекопитающим (классы Cat и Dog), а просто использовать класс Mammal, как показано ниже:

```
class Mammal
{
    MakingNoise speaker;

    public Mammal (MakingNoisesb)
    {
        this.speaker = sb;
    }
    public String makeNoise()
    {
        return this.speaker.makeNoise() ;
    }
}
```

Теперь можно создать экземпляр поведения издания звуков классом Cat и предоставить его классу Animal для создания млекопитающего, поведения которого подобно поведению класса Cat. На самом деле всегда можно собрать класс Cat с помощью внедрения поведений вместо того, чтобы использовать традиционные

```
Mammal cat = new Mammal(new CatNoise());
```

Ниже можно увидеть завершающий шаг — все приложение полностью:

```
public class TestMammal {
    public static void main(String args[]) {

        System.out.printIn("Hello World\n") ;

        Mammal cat = new Mammal (new CatNoise());
        Mammal dog = new Mammal (new DogNoise());

        System.out.printIn("Cat says " + cat.makeNoise());
        System.out.printIn("Dog says " + dog.makeNoise()) ;

    }
}
class Mammal
{
MakingNoise speaker;

    public Mammal (MakingNoisesb)
    {
        this.speaker = sb;
    }
    public String makeNoise()
    {
        return this.speaker.makeNoise() ;
    }
}

interface MakingNoise
{
    public String makeNoise();
}
class CatNoise implements MakingNoise
{
    public String makeNoise()
    {
        return "Meow" ;
    }
}
class DogNoise implements MakingNoise
    {
    public String makeNoise()
    {
        return "Bark";
    }
}
```

При обсуждении внедрения зависимостей важным является то, когда на самом деле создается экземпляр объекта. Несмотря на то что цель состоит в составлении объектов с помощью внедрения, очевидно, в какой-то момент придется создавать экземпляры объектов. В результате принятие решений при проектировании основано на том, в какой момент создавать такой экземпляр.

Как говорилось ранее в этой главе, цель инверсии зависимостей состоит в том, чтобы создать связь класса с чем-то абстрактным, а не конкретным, даже если очевидно, что на каком-то этапе придется создать конкретный объект. Поэтому единственной простой целью ставим создание конкретного объекта (с помощью ключевого слова `new`) вверх по цепочке как можно дальше как, например, в методе `main()`. Всегда оценивайте обстановку, когда видите ключевое слово `new`.

Заключение

На этом мы завершаем обсуждение принципов SOLID. Принципы SOLID — это на сегодняшний день один из самых влиятельных наборов методических рекомендаций для объектно-ориентированного проектирования. Интересно в изучении этих принципов то, как они соотносятся с фундаментальными концепциями объектно-ориентированного проектирования, такими как инкапсуляция, наследование, полиморфизм и композиция, особенно в рамках спора композиции против наследования.

По моему мнению, наиболее интересное, что можно вынести из принципов SOLID, — ничего не нужно урезать и ужимать. Как очевидно из обсуждения проблемы композиции против наследования, даже застарелые фундаментальные концепции объектно-ориентированного программирования открыты для новых интерпретаций. Как мы уже увидели, немного времени наряду с эволюцией различных соответствующих мыслительных процессов приносят инновациям пользу.

Ссылки

Мартин Роберт с соавт. «Гибкая разработка программного обеспечения: принципы, паттерны и практики» (Agile Software Development, Principles, Patterns, and Practices). Бостон: Pearson Education, Inc., 2009

Мартин Р. Чистый код: создание, анализ и рефакторинг. Библиотека программиста. — СПб.: Питер, 2018. — 464 с.: ил.

ОБ ОБЛОЖКЕ

Cover image © SOMRERK WITTHAYANANT/ Shutterstock

Королевский павильон Хо Кхам Луанг находится в самом сердце Королевского парка Раджапрук (Чиангмай, Таиланд). Был построен в 2006 году к 60-й годовщине восхождения на трон Пхумипона Адульядета.

Несущая конструкция выполнена без единого гвоздя, и все сооружение держится на деревянных креплениях. Парящая крыша с многоярусной черепицей является отличительной чертой королевских павильонов древнего тайского королевства Ланна.

Мэтт Вайсфельд

Объектно-ориентированный подход

5-е международное издание

Перевел с английского *И. Сигайлюк*

Заведующая редакцией	*Ю. Сергиенко*
Ведущий редактор	*К. Тульцева*
Литературный редактор	*А. Руденко*
Художественный редактор	*В. Мостипан*
Корректор	*Н. Викторова*
Верстка	*Л. Егорова*

Изготовлено в России. Изготовитель: ООО «Прогресс книга».
Место нахождения и фактический адрес: 194044, Россия, г. Санкт-Петербург,
Б. Сампсониевский пр., д. 29А, пом. 52. Тел.: +78127037373.

Дата изготовления: 12.2019. Наименование: книжная продукция. Срок годности: не ограничен.

Налоговая льгота — общероссийский классификатор продукции ОК 034-2014, 58.11.12 — Книги печатные профессиональные, технические и научные.

Импортер в Беларусь: ООО «ПИТЕР М», 220020, РБ, г. Минск, ул. Тимирязева, д. 121/3, к. 214, тел./факс: 208 80 01.

Подписано в печать 20.12.19. Формат 70×100/16. Бумага офсетная. Усл. п. л. 20,640. Тираж 1000. Заказ 0000.

Отпечатано в ОАО «Первая Образцовая типография». Филиал «Чеховский Печатный Двор».
142300, Московская область, г. Чехов, ул. Полиграфистов, 1.
Сайт: www.chpk.ru. E-mail: marketing@chpk.ru
Факс: 8(496) 726-54-10, телефон: (495) 988-63-87

КНИГА-ПОЧТОЙ

ЗАКАЗАТЬ КНИГИ ИЗДАТЕЛЬСКОГО ДОМА «ПИТЕР» МОЖНО ЛЮБЫМ УДОБНЫМ ДЛЯ ВАС СПОСОБОМ:

- на нашем сайте: **www.piter.com**
- по электронной почте: **books@piter.com**
- по телефону: **(812) 703-73-74**

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ОПЛАТЫ:

- Наложенным платежом с оплатой при получении в ближайшем почтовом отделении.
- С помощью банковской карты. Во время заказа вы будете перенаправлены на защищенный сервер нашего оператора, где сможете ввести свои данные для оплаты.
- Электронными деньгами. Мы принимаем к оплате Яндекс.Деньги, Webmoney и Qiwi-кошелек.
- В любом банке, распечатав квитанцию, которая формируется автоматически после совершения вами заказа.

ВЫ МОЖЕТЕ ВЫБРАТЬ ЛЮБОЙ УДОБНЫЙ ДЛЯ ВАС СПОСОБ ДОСТАВКИ:

- Посылки отправляются через «Почту России». Отработанная система позволяет нам организовывать доставку ваших покупок максимально быстро. Дату отправления вашей покупки и дату доставки вам сообщат по e-mail.
- Вы можете оформить курьерскую доставку своего заказа (более подробную информацию можно получить на нашем сайте www.piter.com).
- Можно оформить доставку заказа через почтоматы,(адреса почтоматов можно узнать на нашем сайте www.piter.com).

ПРИ ОФОРМЛЕНИИ ЗАКАЗА УКАЖИТЕ:

- фамилию, имя, отчество, телефон, e-mail;
- почтовый индекс, регион, район, населенный пункт, улицу, дом, корпус, квартиру;
- название книги, автора, количество заказываемых экземпляров.

БЕСПЛАТНАЯ ДОСТАВКА:

- курьером по Москве и Санкт-Петербургу при заказе на сумму **от 2000 руб.**
- почтой России при предварительной оплате заказа на сумму **от 2000 руб.**