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.

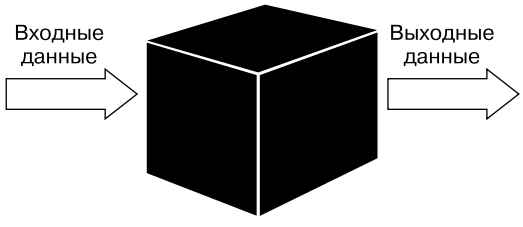

Рис. 1.1. Черный ящик

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.

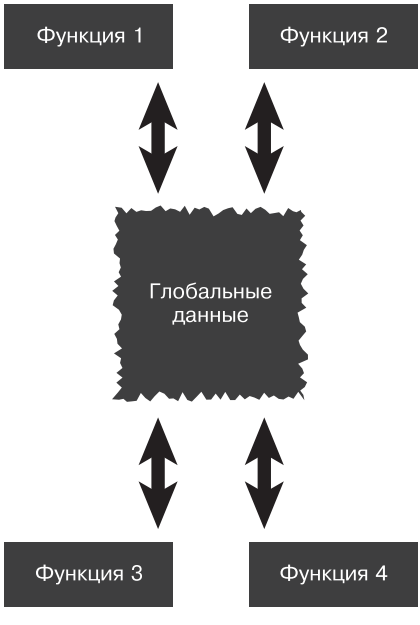

Рис. 1.2. Использование глобальных данных

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.

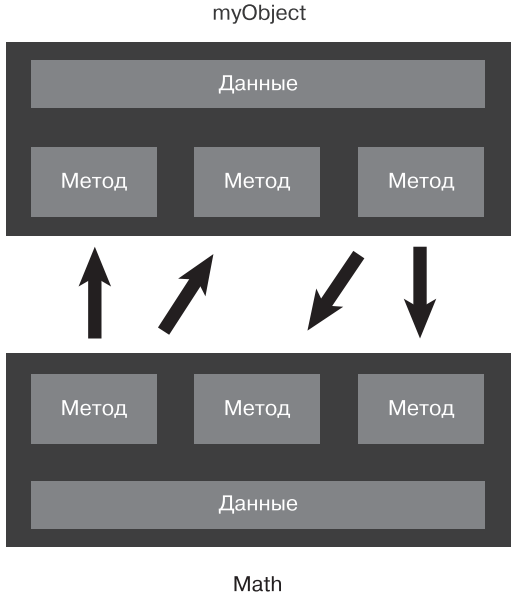

Рис. 1.3. Коммуникации между объектами

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.

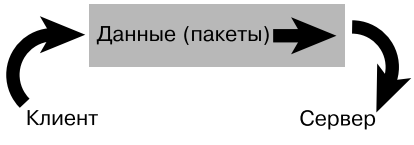

Рис. 1.4. Данные, передаваемые по сети

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.

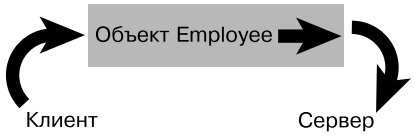

Рис. 1.5. Объект, передаваемый по сети

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

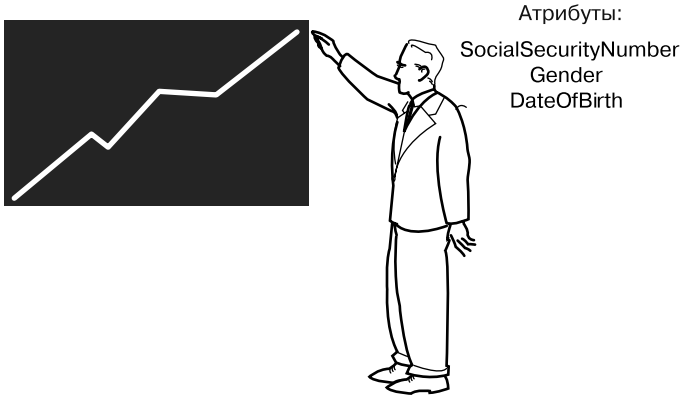

Рис. 1.6. Атрибуты объекта Employee

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».

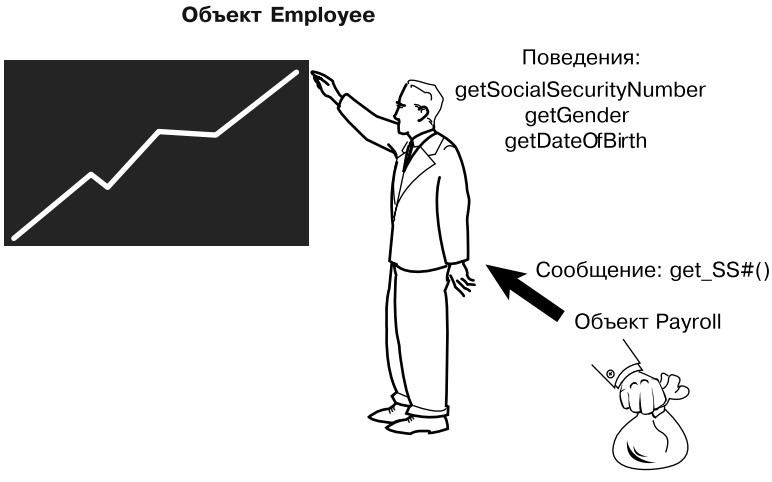

Рис. 1.7. Поведения объекта Employee

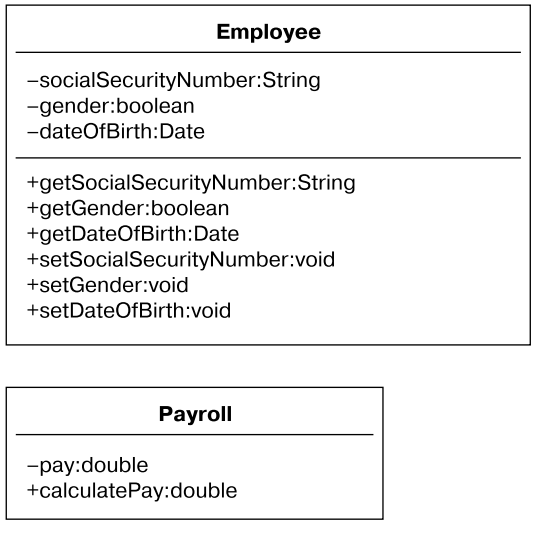

Рис. 1.8. Диаграммы классов Employee и Payroll

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

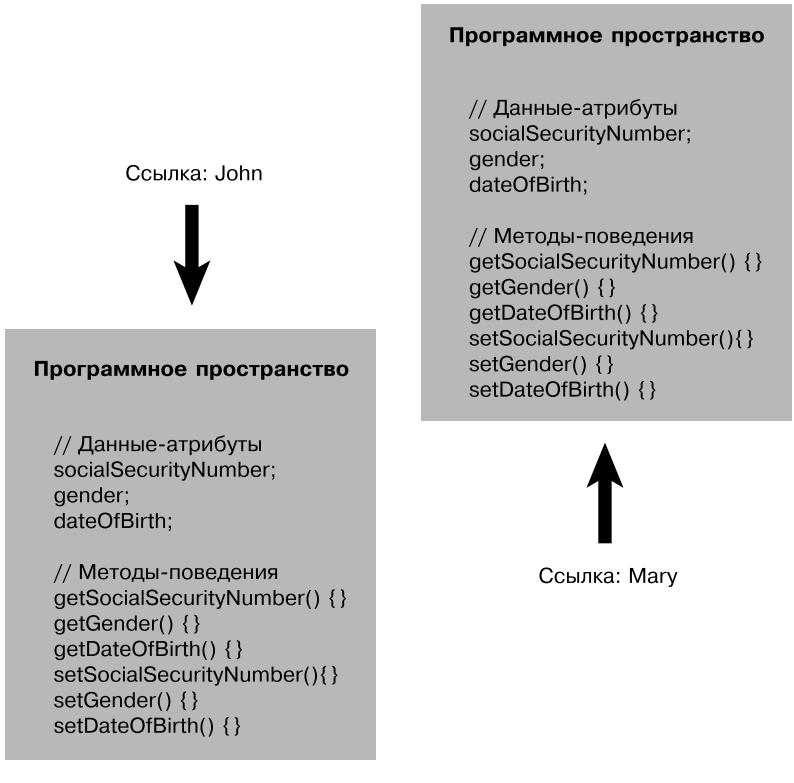

Рис. 1.9. Программные пространства

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.

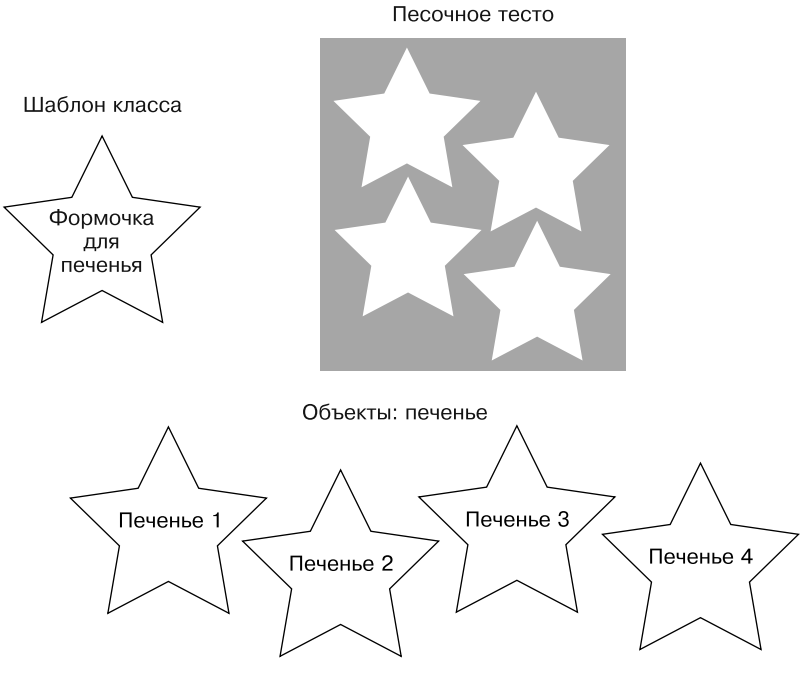

Рис. 1.10. Шаблон класса

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.

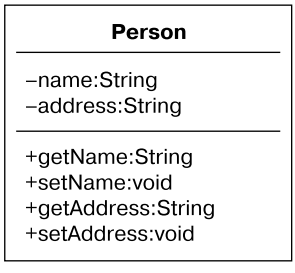

Рис. 1.11. Диаграмма класса Person

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.

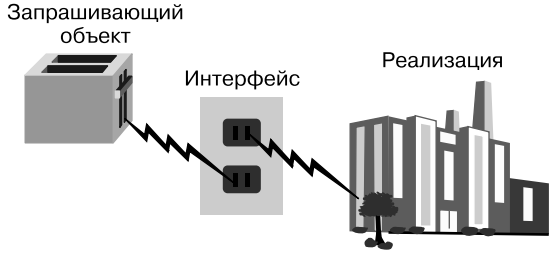

Рис. 1.12. Пример с электростанцией

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.

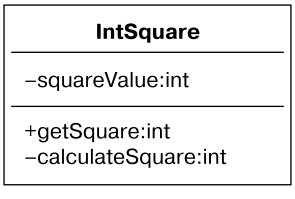

Рис. 1.13. Класс IntSquare

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
```

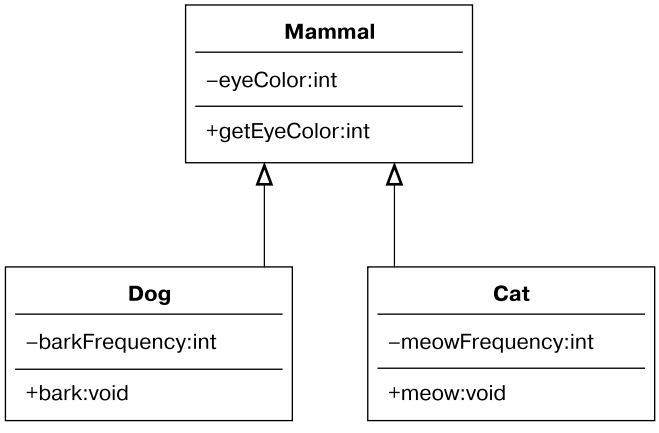

Рис. 1.14. Иерархия классов млекопитающих

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.

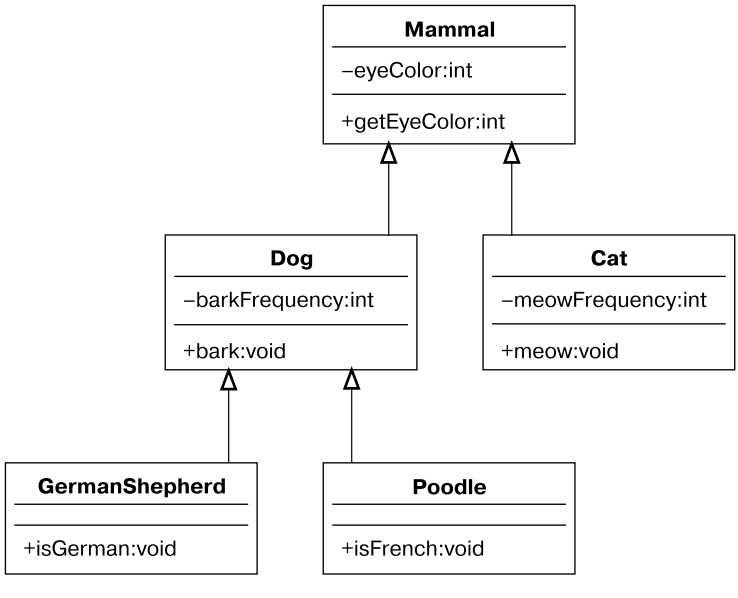

Рис. 1.15. UML-диаграмма классов млекопитающих

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).

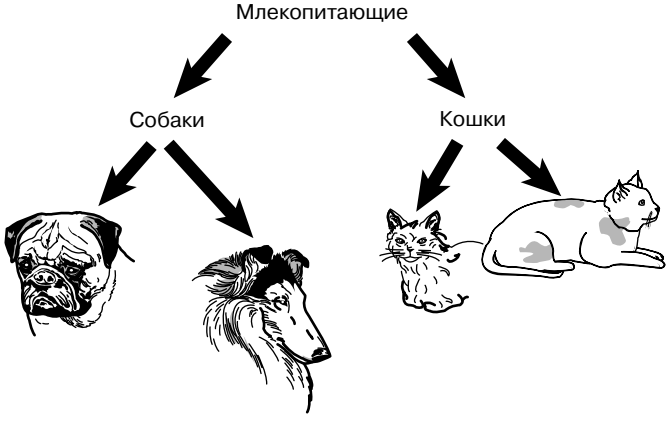

Рис. 1.16. Иерархия млекопитающих

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.

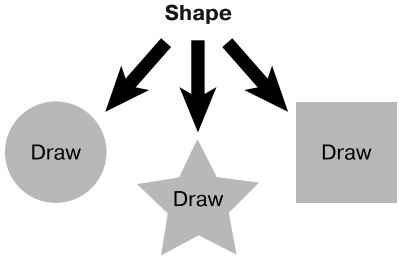

Рис. 1.17. Иерархия Shape

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);
```

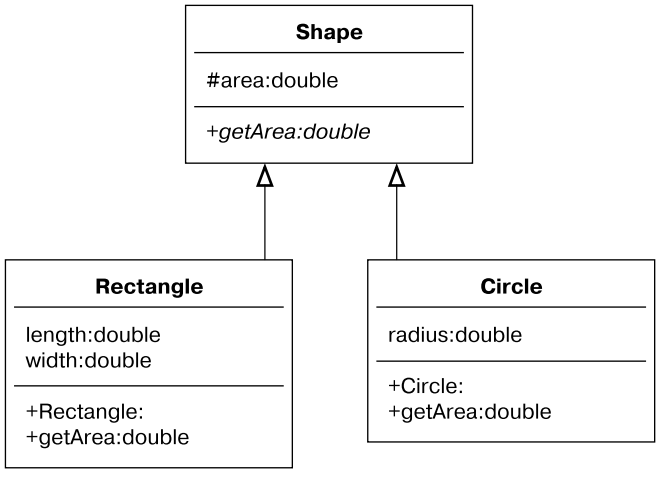

Рис. 1.18. UML-диаграмма Shape

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.