O'REILLY®

ГИД JAVA-разработчика

Проектно-ориентированный подход

Рауль-Габриэль Урма
Ричард Уорбертон

БОМБОРА

МИРОВОЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

O'REILLY®

Raoul-Gabriel Urma
Richard Warburton

Real-World Software Development

A Project-Driven Guide to Fundamentals in Java

O'REILLY®

Рауль-Габриэль Урма
Ричард Уорбертон

ГИД JAVA разработчика

Проектно-ориентированный подход

БОМБОРА
ИЗДАТЕЛЬСТВО
Москва 2022

УДК 004.43
ББК 32.973.26-018.1
У69

Real-World Software Development
Raoul-Gabriel Urma and Richard Warburton
© 2021 Eksmo Publishing Company Authorized Russian translation of the English edition of Real-World Software Development ISBN 9781491967171
© 2020 Functor Ltd. and Monotonic Ltd. This translation is published and sold by permission of O'Reilly Media, Inc., which owns or controls all rights to publish and sell the same.

Урма, Рауль-Габриэль.

У69 Гид Java-разработчика : проектно-ориентированный подход / Рауль-Габриэль Урма, Ричард Уорбертон ; [перевод с английского М. А. Райтман]. — Москва : Эксмо, 2022. — 224 с. : ил. — (Мировой компьютерный бестселлер).

ISBN 978-5-04-094955-7

На примере реальных проектов авторы разбирают все наиболее популярные приемы объектно-ориентированного программирования, такие как разработка через тестирование или функциональное программирование. В этом руководстве представлен проектно-ориентированный подход к разработке программного обеспечения на языке Java, позволяющий освоить ключевые навыки, необходимые каждому эффективному программисту.

УДК 004.43
ББК 32.973.26-018.1

ISBN 978-5-04-094955-7

© Райтман М.А., перевод на русский язык, 2022
© Оформление. ООО «Издательство «Эксмо», 2022

https://t.me/it_boooks

Предисловие

Совершенствование навыков разработки программного обеспечения включает в себя изучение целого пула разрозненных концепций. Если вы — начинающий разработчик или даже имеете определенный опыт, это может показаться непреодолимо сложной задачей. Стоит ли потратить время на изучение основных положений объектно-ориентированного мира, таких как принципы SOLID, шаблоны проектирования или разработка через тестирование? Нужно ли применять такие набирающие популярность технологии, как, например, функциональное программирование?

Часто бывает, что, даже, изучив какие-то темы в отдельности, потом трудно понять, как связать их друг с другом. В каком случае необходимо воспользоваться в проекте функциональным программированием? Когда стоит задуматься о тестировании? Как узнать, в какой момент следует внедрять или совершенствовать упомянутые методы? Нужно ли прочитать книгу по каждой из этих тем, затем изучить посвященные им блоги, или посмотреть видео, чтобы разобраться, как все это соединить воедино? И вообще, с чего начать?

Не переживайте. Данная книга создана для того, чтобы помочь вам. Вы получите ответы на свои вопросы в процессе интегрированного, проектно-ориентированного обучения. Вы изучите основные темы, которые помогут вам стать эффективным разработчиком. Кроме того, мы покажем, как объединить полученные знания и применить в больших проектах.

Почему мы написали эту книгу

За многие годы обучения программированию мы накопили богатый опыт. Мы оба написали книги по Java 8 и проводим курсы по профессиональной разработке программного обеспечения. В процессе мы были признаны Java-

чемпионами и стали востребованными спикерами на различных международных конференциях.

За все это время мы поняли, что многим разработчикам достаточно лишь введения или краткого описания некоторых основных тем. Шаблоны проектирования, функциональное программирование, принципы SOLID и тестирование — это методы, которые хороши сами по себе, но редко можно встретить демонстрацию того, как соединить их вместе. Иногда программисты перестают развивать свои навыки лишь потому, что не могут определиться, что же изучать дальше. Мы хотим не просто дать вам определенные базовые навыки, наша цель — сделать обучение легким, интересным и даже веселым.

Подход, ориентированный на разработчика

Данная книга предоставляет возможность познакомиться с ориентированным на разработчика подходом. В ней вы найдете множество примеров кода. В каждой теме рассматриваются реальные программы и проекты. Все они даются в полном объеме, так что на каждом этапе вы можете проверять код в своей *интегрированной среде разработки* (*Integrated Development Environment*, *IDE*) и запускать программы, чтобы оценить их в действии.

Есть и другая распространенная проблема технической литературы. Зачастую она написана в формальном стиле учебника, далеком от живого общения. Однако мы решили придерживаться в книге разговорного стиля, чтобы помочь вам почувствовать себя членом команды, а не «подопечным» или учеником.

Что в этой книге?

Каждая глава книги строится вокруг определенного проекта. По завершении главы вы сможете самостоятельно написать этот проект, если, конечно, хорошо изучите весь материал. При этом уровень сложности будет постепенно возрастать — от простых консольных программ до полноценных приложений.

Пректно-ориентированный подход, лежащий в основе данной книги, имеет ряд преимуществ. Прежде всего, он позволит вам увидеть, как разнообразные методики программирования работают в единой связке. Когда, ближе

к концу книги, мы начнем рассматривать функциональное программирование, речь пойдет не просто о наборе абстрактных вычислительных операций. Они будут нужны для получения определенных результатов в конкретном проекте. Такой подход выгодно отличает данное руководство от учебных материалов, демонстрирующих действительно хорошие методы, но в отрыве от реальных задач, из-за чего разработчики часто применяют их совершенно неуместно.

Проектно-ориентированный подход позволит вам на каждом этапе работать с реальными примерами. В классических учебных пособиях довольно часто можно встретить в коде метапеременные `Foo` (для обозначения классов) и `bar` (для обозначения методов). Наши примеры соответствуют реальным задачам и демонстрируют подход к решению реальных проблем, похожих на те, с которыми, возможно, вы столкнетесь в повседневной работе.

И наконец, проектно-ориентированный подход делает учебу гораздо интереснее. Каждая глава — это новый проект и новая возможность совершить для себя открытие. Мы хотим, чтобы вы дочитали все до конца, и искренне рады каждому читателю. Все главы книги начинаются с той или иной задачи, которую нужно решить. Затем рассматривается решение, а в конце подводятся итоги по пройденному материалу. Мы четко определяем задачу в начале и в конце каждой главы, чтобы убедиться, что вы хорошо ее понимаете.

Для кого эта книга?

Мы уверены, что разработчики из самых разных областей найдут для себя что-то полезное и интересное в этой книге. Среди них будут и те, кто сможет использовать ее максимально эффективно.

На наш взгляд, основная аудитория книги — это начинающие разработчики, только окончившие университет, либо имеющие за плечами пару лет опыта работы. Мы познакомим вас с основными темами, которые, как мы считаем, пригодятся вам в работе. Для взаимодействия с данной книгой не нужно иметь ученую степень, но для лучшего усвоения материала необходимо обладать базовыми знаниями в программировании. К примеру, здесь мы не будем объяснять, что такое оператор `if` или циклы.

Для того чтобы приступить, вам необязательно иметь глубокие познания в объектно-ориентированном или функциональном программировании.

Например, в главе 2 от вас не потребуется ничего сверх знания о том, что такое класс, и умения работать с базовыми типами (такими как `List<String>`). Только самые основы.

Данная книга может также представлять интерес и для разработчиков, пришедших в Java из других языков программирования, таких как C#, C++ или Python. Она поможет вам быстро освоить необходимые конструкции языка, принципы, методы и ключевые моменты, необходимые для написания хорошего кода на Java.

Если вы являетесь более опытным Java-разработчиком, то вполне можете пропустить главу 2, чтобы не повторять базовый материал, который вы и так уже знаете. Однако начиная с главы 3 в книге будут рассматриваться многие концепции и подходы, полезные для всех разработчиков.

Мы считаем, что обучение может быть одним из самых интересных этапов разработки программного обеспечения, и верим, что во время чтения книги вы с нами согласитесь. Надеемся, вам понравится это путешествие.

Условные обозначения, используемые в книге

В книге используются следующие типографские условные обозначения:

Курсив

Обозначает новые понятия и термины, URL ссылки, адреса электронной почты, имена файлов, расширения файлов.

`Моноширинный шрифт`

Используется в листингах программ, а также в тексте книги для обозначения элементов программ, таких как имена переменных или функций, базы данных, типы данных, переменные среды, операторы и ключевые слова.

`Моноширинный полужирный`

Обозначает команды или другой текст, который должен быть введен пользователем.

`Моноширинный курсивный`

Обозначает текст, который должен быть заменен пользовательскими значениями или значениями, уточняемыми в контексте.


Данный символ обозначает примечание.

Использование примеров кода

Дополнительные материалы (такие как примеры программ, упражнения и т. д.) доступны для скачивания на *https://github.com/Iteratr-Learning/Real-World-Software-Development*.

Данная книга создана для того, чтобы помочь вам в работе. Поэтому, если вам подходят некоторые примеры программ, вы можете использовать их в своих приложениях и в документации. Вам не нужно связываться с нами и получать для этого специальное разрешение, оно может понадобиться только при использовании внушительного объема кода. Например, если в своей программе вы используете несколько кусочков кода из этой книги, разрешение вам не нужно. Использование большого объема примеров кода в документации к вашему продукту требует получения разрешения.

Начало путешествия

В этой главе мы познакомим вас с устройством книги и правилами работы с ней. В целом основной подход к материалу в книге можно описать следующим образом: *практика и общие принципы выше конкретной технологии*. На сегодняшний день существует множество книг по каким-то конкретным направлениям, и мы не стремимся стать частью этого огромного собрания. Мы не хотим сказать, что узкопрофильные знания по какому-то конкретному языку, среде или библиотеке не будут полезны. Просто их жизненный цикл короче, чем у общих знаний и понятий, которые можно применять к разным языкам программирования и технологиям в течение длительного периода времени. Вот в чем вам должна помочь эта книга.

Темы

Мы использовали в книге проектно-ориентированную структуру, чтобы вам было удобнее учиться. Такой подход позволяет объединить разные темы, заставляет задуматься о том, как они связаны друг с другом и почему мы их выбрали. Ниже перечислены четыре темы, которые переплетаются в последующих главах.

Особенности Java

Структурирование кода при помощи классов и интерфейсов обсуждается в главе 2. Исключения и пакеты рассматриваются в главе 3, из которой вы также узнаете о лямбда-выражениях. Глава 5 познакомит вас с выводом типа переменной и `switch`-выражениями. И, наконец, в главе 7

уже более детально рассматриваются лямбда-выражения и ссылки на методы.

Большое количество программных продуктов написаны на Java, поэтому очень важно знать особенности этого языка и понимать, как он работает. Многие из этих особенностей характерны и для других языков программирования, таких как C#, C++, Ruby или Python. Несмотря на то что перечисленные языки имеют различия, понимание того, как использовать классы и основные принципы ООП, будут полезны при работе с любым из них.

Разработка программного обеспечения и архитектура

По ходу книги вы познакомитесь с рядом шаблонов проектирования, которые помогут вам находить стандартные решения распространенных проблем, встречающихся в разработке. Это важно, поскольку, несмотря на то, что каждый проект является индивидуальным и решает конкретные задачи, на самом деле многие из этих задач уже встречались ранее. Понимание стандартных проблем и существующих методов их решения, придуманных другими программистами, убережет вас от изобретения колеса и позволит существенно сэкономить время на разработке.

В главе 2 представлены концепции связанности и связности. Шаблон уведомления рассматривается в главе 3. Как построить «дружелюбный» Fluent API и шаблон Builder, описано в главе 5. В главе 6 рассматривается концепция «целостной картины» событийно-ориентированной и гексагональной архитектур, а в главе 7 представлен шаблон Repository. Наконец, также в главе 7 вы познакомитесь с функциональным программированием.

SOLID

В разных главах книги мы рассмотрим все принципы SOLID, представляющие собой набор правил и наилучших подходов, разработанных с целью облегчения обслуживания программного обеспечения. Если программа, которую вы написали, стала успешной, она потребует развития и сопровождения. Попытки сделать программное обеспечение максимально простым в обслуживании способствуют его «эволюции», добавлению возможностей и функций на долгосрочную перспективу.

Итак, принципы SOLID и главы, в которых они рассматриваются:

- Принцип единственной ответственности (SRP — Single Resposibility Principle) — глава 2.
- Принцип открытости/закрытости (OCP — Open/Closed Principle) — глава 3.
- Принцип подстановки Барбары Лисков (LSP — Liskov Substitution Principle) — глава 4.
- Принцип разделения интерфейса (ISP — Interface Segregation Principle) — глава 5.
- Принцип инверсии зависимостей (DIP — Dependency Inversion Principle) — глава 7.

Тестирование

Написание надежного кода, который со временем можно легко доработать, очень важно. Ключ к этому — автоматизированное тестирование. По мере того как разрастается ваша программа, становится все сложнее тестировать всевозможные случаи вручную. Необходимо автоматизировать процесс тестирования, чтобы избежать многодневного человеческого труда.

Базовые знания о создании тестов вы получите в главах 2 и 4. В главе 5 мы поговорим о разработке через тестирование (TDD — Test-driven Development). И наконец, в главе 6 мы рассмотрим применение «тестовых двойников» (дублеров), включая Mocks и Stubs.

Структура глав

Краткое содержание глав.

Глава 2. Анализатор банковских операций

Вы напишете программу для анализа банковских операций, чтобы помочь людям лучше разбираться в своих финансах. Вы изучите основные принципы объектно-ориентированного программирования, такие как принцип единственной ответственности (SRP), связанность и связность.

Глава 3. Расширяем анализатор банковских операций

В этой главе вы узнаете, как можно доработать код, написанный в главе 2, добавив дополнительные возможности, использовав шаблон Strategy Design,

принцип открытости/закрытости, а также научитесь перехватывать ошибки при помощи исключений.

Глава 4. Система управления документами

В этой главе мы поможем одному успешному доктору систематизировать карточки пациентов. Вы познакомитесь с наследованием, с принципом подстановки Барбары Лисков и компромиссом между созданием и наследованием. Вы также узнаете, как создавать более надежное программное обеспечение за счет применения автоматизированного тестирования.

Глава 5. Движок бизнес-правил

Вы узнаете о создании базового механизма бизнес-правил — гибкого и простого в обслуживании способа определения бизнес-логики. Эта глава познакомит вас с разработкой через тестирование, разработкой Fluent API и принципом разделения интерфейсов.

Глава 6. Twootr

Twootr — это платформа, позволяющая пользователям делиться короткими сообщениями со своими подписчиками. В данном случае демонстрируется создание простой системы Twootr. Вы научитесь «смотреть извне» — то есть двигаться от требований к приложению к его ядру. Вы также узнаете, как применять «тестовые двойники» для тестирования взаимодействия различных компонентов вашего кода.

Глава 7. Расширяем Twootr

В последней главе, посвященной проектам, мы займемся развитием платформы Twootr. В этой главе объясняется принцип инверсии зависимостей, а также представлены архитектурные решения «целостной картины» вроде событийно-ориентированной или гексагональной архитектуры. Вы сможете расширить свои знания в области автоматизированного тестирования за счет изучения «тестовых двойников» и техник функционального программирования.

Глава 8. Заключение

В финальной главе кратко рассматриваются основные темы и концепции, изложенные в книге. Читателю предлагаются дополнительные ресурсы, которые могут пригодиться ему в дальнейшей работе.

Самостоятельная работа

Будучи разработчиком, вы должны подходить к проектам итеративно. Что это значит? Выберите для себя одну или две наиболее приоритетные задачи, реализуйте их, а затем, используя обратную связь и отзывы, определите следующие задачи. На наш взгляд, это отличный способ оценить свои навыки.

В конце каждой главы вы найдете небольшой раздел под названием «Самостоятельная работа», содержащий ряд предложений, которые могут помочь вам лучше усвоить пройденный материал.

Теперь, когда вы знаете, чего ожидать от этой книги, приступим к работе!

Анализатор банковских операций

Задача

В компании FinTech сейчас реально жарко. Марк Эрбергцук понимает, что тратит много денег на различные закупки, и хочет, чтобы все его расходы подсчитывались автоматически. Конечно, он получает ежемесячные отчеты из банка, но они его не устраивают. Поэтому он просит вас разработать программный продукт, который мог бы автоматически вести учет всех его банковских операций, чтобы иметь возможность лучше контролировать свои финансы. Задание принято!

Цель

В этой главе вы познакомитесь с основами «правильной» разработки программного обеспечения, прежде чем пойти дальше и приступить к изучению более продвинутых технологий в следующих главах.

Вы начнете с реализации решения задачи при помощи одного класса. Затем узнаете, в чем заключаются недостатки такого подхода с точки зрения адаптации к меняющимся требованиям и обслуживания проекта.

Но не переживайте! Вы освоите принципы и техники разработки, которые помогут вам убедиться, что ваш код соответствует указанным критериям. Прежде всего, вы узнаете о *принципе единственной ответственности* (SRP), который позволяет разрабатывать более удобное в обслуживании и более «прозрачное» программное обеспечение, а также снижает вероятность возникновения новых багов. По пути вы также познакомитесь с такими концепциями, как *связность* и *связанность*, которые помогут вам оценить качество кода и разрабатываемого вами программного продукта.

В этой главе мы пользуемся библиотеками и свойствами Java версии 8 и выше, включая новые библиотеки для работы с датами и временем. Если в какой-то момент вы захотите посмотреть исходный код, вы можете найти его в репозитории книги `/com/iteratrlearning/shu_book/chapter_02`.

Требования к анализатору банковских операций

Вы встретились с Марком Эрбергцуком, чтобы обсудить требования к программе и выпить по чашечке настоящего хипстерского латте (без сахара). Поскольку Марк технически подкован, он объяснил вам, что анализатор банковских операций должен просто считывать текстовый файл, в котором содержится список транзакций. Он заранее скачал этот файл из своего интернет-банкинга. Файл выполнен в виде значений, разделенных запятыми (Comma-Separated Values — CSV формат). Вот пример транзакций:

```
30-01-2017,-100,Deliveroo

30-01-2017,-50,Tesco
01-02-2017,6000,Salary
02-02-2017,2000,Royalties
02-02-2017,-4000,Rent
03-02-2017,3000,Tesco
05-02-2017,-30,Cinema
```

Марк пояснил, что хочет получить ответы на следующие вопросы:

- Какова общая сумма начислений и списаний по списку операций? Она отрицательная или положительная?
- Сколько транзакций было в конкретном месяце?
- 10 самых затратных операций.
- На что было потрачено больше всего денег?

Принцип KISS

Начнем с простого, а именно — с первого вопроса: «Какова общая сумма начислений и списаний по списку операций?» Итак, вам нужно обработать CSV-файл и посчитать сумму всех операций. Поскольку больше ничего не

требуется, вероятно, вы решите, что не имеет смысла создавать очень сложное приложение.

Вы можете «сделать его коротким и простым» (Keep It Short and Simple — KISS), уместив все приложение в одном классе, как показано в примере 2-1. Обратите внимание, что пока вам не нужно задумываться об обработке исключений (например, если файл не существует или если возникла ошибка при считывании файла). С этой темой вы познакомитесь в главе 3.

Формат CSV не полностью стандартизирован. Чаще всего для разделения значений в нем используются запятые. Однако многие считают, что в качестве разделителя могут выступать также и другие символы, такие как точка с запятой или символ табуляции. Данные особенности способны привести к усложнению программы-парсера. Поэтому в этой главе мы будем считать, что значения разделяются только запятыми (,).

Пример 2-1. Вычисление суммы операций

```
public class BankTransactionAnalyzerSimple {
    private static final String RESOURCES = "src/main/resources/";

    public static void main(final String... args) throws IOException {

        final Path path = Paths.get(RESOURCES + args[0]);
        final List<String> lines = Files.readAllLines(path);
        double total = 0d;
        for(final String line: lines) {
            final String[] columns = line.split(",");
            final double amount = Double.parseDouble(columns[1]);
            total += amount;
        }

        System.out.println("The total for all transactions is " + total);
    }
}
```

Итак, что происходит? Вы загружаете CSV-файл, переданный приложению в виде аргумента командной строки. Класс `Path` представляет собой путь в файловой системе. Затем вы используете метод `Files.readAllLines()`, который возвращает список строк. После прочтения всех строк файла можно приступить к их разделению:

- Разделяем столбцы при помощи запятых.
- Извлекаем сумму.
- Преобразуем сумму в тип `double`.

После получения суммы в виде `double` для данной конкретной операции можно добавить ее к общей сумме. В конце вычислений вы получаете итоговую сумму.

Код из примера 2-1 вполне рабочий, однако в нем упущены некоторые ключевые моменты, о которых всегда стоит помнить при написании кода:

- Что, если файл окажется пустым?
- Что, если не получится посчитать сумму из файла из-за поврежденных данных?
- Что, если в строке пропущены какие-либо данные?

Мы еще вернемся к этим пунктам, когда будем разбирать исключения в главе 3, но, несомненно, подобные вопросы нужно держать в голове всегда.

Что насчет решения второй задачи: «Сколько транзакций было в конкретном месяце?» Что мы можем сделать? Самое простое решение — скопировать и вставить, верно? Вы можете просто скопировать и вставить тот же код, изменив логику так, чтобы он выбирал нужный месяц, как показано в примере 2-2.

Пример 2-2. Вычисление суммы за январь

```
final Path path = Paths.get(RESOURCES + args[0]);
final List<String> lines = Files.readAllLines(path);
double total = 0d;
final DateTimeFormatter DATE_PATTERN = DateTimeFormatter.ofPattern("dd-MM-yyyy");
for(final String line: lines) {
    final String[] columns = line.split(",");
    final LocalDate date = LocalDate.parse(columns[0], DATE_PATTERN);
    if(date.getMonth() == Month.JANUARY) {
        final double amount = Double.parseDouble(columns[1]);
        total += amount;
    }
}

System.out.println("The total for all transactions in January is " + total);
```

Переменные final

В качестве краткого экскурса мы объясним использование ключевого слова `final` в наших примерах, тем более что мы будем пользоваться им довольно часто на протяжении всей книги. Пометка `final` у поля или переменной означает, что они не могут быть повторно определены. Использовать `final` или нет — решение исключительно ваше и вашей команды, поскольку оно имеет как положительные, так и отрицательные стороны. Мы для себя установили, что, помечая ключевым словом `final` как можно больше переменных, мы четко определяем, какое состояние жестко зафиксировано, а какое может изменяться в процессе жизни объекта.

С другой стороны, использование `final` не может однозначно гарантировать неизменность объекта. Например, поле `final` может *ссылаться* на изменяемый объект. Подробнее о неизменяемости или иммутабельности мы поговорим в главе 4. Забегая вперед, можно сказать, что применение `final` привносит в код больше шаблонности. Некоторые команды разработчиков прибегают к компромиссной позиции, используя `final` в параметрах методов, чтобы убедиться, что они точно не переопределены и не являются локальными переменными.

Также не имеет особого смысла использовать ключевое слово `final`, хотя Java это допускает, в параметрах абстрактных методов. Например, в интерфейсах. Причина — в отсутствии тела метода. Вероятно, использование `final` сильно сократилось после введения ключевого слова `var` в Java 10. Мы обсудим это позже в примере 5-15.

Обслуживаемость кода и антишаблоны

Как вы считаете, «копипаст», к которому мы прибегли в примере 2-2, — это хороший подход? Настало время сделать шаг назад и подумать об этом. Когда вы программируете, ваша задача — постараться сделать код хорошо *обслуживаемым*. Что это значит? Лучше всего объяснить это при помощи списка свойств, которыми должен обладать ваш код.

- Должно быть легко находить участок кода, отвечающий за определенную функцию.
- Должно быть понятно, как код работает.
- Должно быть просто убрать или добавить новую опцию.

- Должна обеспечиваться *инкапсуляция*. Другими словами, реализация деталей должна быть скрыта от пользователя таким образом, чтобы ему было легко разобраться и внести изменения.

Хороший способ оценить эффективность кода — это представить, что вы ушли в другую компанию, предоставив разбираться с вашим кодом коллеге.

Ваша первоочередная задача — контролировать сложность приложения, которое вы создаете. При этом, если вы будете продолжать «копипастить» одни и те же участки кода по мере добавления новых возможностей, то в конце концов, вы столкнетесь со следующими проблемами, которые называются *антишаблонами*, или *антипаттернами*, потому что являются наиболее неэффективными решениями:

- Тяжелый для понимания код, потому что у вас есть один огромный «*класс-бог*».
- Хрупкий код, который разваливается от любых изменений из-за многочисленного *дублирования*.

Давайте рассмотрим эти два антишаблона более подробно.

Класс-бог

Поместив весь свой код в одном файле, вы придете к одному огромному классу, с которым невероятно сложно разобраться, потому что он делает абсолютно все! Если вам понадобится обновить логику существующего кода (например, изменить алгоритм синтаксического анализа — парсинга — слов), вы не сможете оперативно найти нужный участок кода и внести изменения. Данная проблема вызвана использованием антишаблона под названием «класс-бог». По сути, у вас есть класс, который делает все. Такой практики, безусловно, следует избегать. В следующем разделе вы познакомитесь с *принципом единственной ответственности*, который является «проводником» в разработке простого для понимания и обслуживаемого кода.

Дублирование кода

Для каждой задачи вы дублируете алгоритм чтения и анализа входных данных. А что если формат входных данных изменится с CSV на JSON? Что если понадобится поддержка нескольких форматов? Добавление этой опции

станет сущим адом, так как ваш код перегружен одним и тем же решением, продублированным много раз в разных местах. Таким образом, вам придется вносить изменения везде, повышая вероятность возникновения багов.

Вы будете часто слышать о принципе «Не повторяйся» (DRY — Don't Repeat Yourself). Его идея заключается в том, что, если успешно избегать дублирования кода, внесение изменений в логику приложения не потребует многочисленных изменений в коде.

Похожая проблема может возникнуть, если вам понадобится изменить формат данных. Сейчас программа способна работать только с конкретным шаблоном формата данных. Если его нужно расширить (к примеру, добавить новые колонки) или ввести поддержку другого формата данных (например, другие названия атрибутов), вам снова придется вносить большое количество изменений по всему коду.

В заключение можно сказать, что лучше делать все максимально просто (по возможности), при этом не стоит злоупотреблять принципом KISS. Вам стоит поразмыслить над проектом всего приложения в целом и понять, как разбить задачу на несколько подзадач, которые проще решать в отдельности. В результате вы получите простой, понятный, удобный в обслуживании и готовый к изменениям код.

Принцип единственной ответственности

Принцип единственной ответственности (SRP) — главный принцип, которого стоит придерживаться при разработке программного обеспечения, если вы хотите создавать простой и удобный в обслуживании код.

Данный принцип можно охарактеризовать при помощи двух утверждений:

- Один класс несет ответственность за одну функцию приложения.
- Никогда не должно быть больше одной причины для внесения изменений в класс[1].

Обычно SRP применяется для классов и методов. Он связан с одной конкретной концепцией, категорией или с каким-то определенным поведением. Благодаря данному принципу создается более устойчивый код, поскольку

[1] Это определение введено Робертом Мартином. — *Прим. авт.*

он может быть изменен только по одной конкретной причине. Почему много причин — это плохо? Ответ вам уже знаком. Как вы успели убедиться ранее, такой подход усложняет код и ухудшает его обслуживаемость за счет потенциально возможного возникновения багов в разных местах. Он также затрудняет понимание кода и возможность внесения в него изменений.

Как же нам применить SRP в коде, приведенном в примере 2-2? Очевидно, что главный класс выполняет сразу несколько задач, которые можно разбить на подзадачи:

1. Считывание входных данных.
2. Синтаксический анализ данных в заданном формате.
3. Обработка результата.
4. Выдача результатов суммирования.

В этой главе мы сфокусируем свое внимание на синтаксическом анализе данных. В следующей главе рассмотрим, как модернизировать анализатор банковских операций таким образом, чтобы приложение стало полностью модульным.

Вернемся к нашему примеру. В первую очередь необходимо создать отдельный класс, предназначенный для извлечения данных из CSV-файла. Таким образом, вы сможете многократно использовать его в различных задачах. Давайте назовем его `BankStatementCSVParser`, чтобы по названию сразу понимать его предназначение (пример 2-3).

Пример 2-3. Помещаем алгоритм парсинга в отдельный класс

```
public class BankStatementCSVParser {

    private static final DateTimeFormatter DATE_PATTERN
        = DateTimeFormatter.ofPattern("dd-MM-yyyy");

    private BankTransaction parseFromCSV(final String line) {
        final String[] columns = line.split(",");

        final LocalDate date = LocalDate.parse(columns[0], DATE_PATTERN);
        final double amount = Double.parseDouble(columns[1]);
        final String description = columns[2];

        return new BankTransaction(date, amount, description);
    }
```

```
    public List<BankTransaction> parseLinesFromCSV(final List<String> lines) {
        final List<BankTransaction> bankTransactions = new ArrayList<>();
        for(final String line: lines) {
            bankTransactions.add(parseFromCSV(line));
        }
        return bankTransactions;
    }
}
```

Как видите, класс `BankStatementCSVParser` объявляет два метода: `parseFromCSV()` и `parseLinesFromCSV()`, которые создают объекты `BankTransaction`, являющиеся экземплярами доменного класса, моделирующего банковские операции (смотрите его объявление в примере 2-4).

Что значит *доменный*? Это означает использование слов и терминологии, соответствующих решаемой бизнес-задаче.

Класс `BankTransaction` очень полезен, так как различные части нашего приложения могут иметь одинаковое представление о том, что такое банковская операция. Класс реализует методы `equals` и `hashcode`. Назначение этих методов и правила их использования описаны в главе 6.

Пример 2-4. Доменный класс для банковских операций

```
public class BankTransaction {
    private final LocalDate date;
    private final double amount;
    private final String description;

    public BankTransaction(final LocalDate date, final double amount, final String
description) {
        this.date = date;
        this.amount = amount;
        this.description = description;
    }

    public LocalDate getDate() {
        return date;
    }
```

```
    public double getAmount() {
        return amount;
    }

    public String getDescription() {
        return description;
    }

    @Override
    public String toString() {
        return "BankTransaction{" +
                "date=" + date +
                ", amount=" + amount +
                ", description='" + description + '\'' +
                '}';
    }

    @Override
    public boolean equals(Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
        BankTransaction that = (BankTransaction) o;
        return Double.compare(that.amount, amount) == 0 &&
                date.equals(that.date) &&
                description.equals(that.description);
    }

    @Override
    public int hashCode() {
        return Objects.hash(date, amount, description);
    }
}
```

Теперь вы можете изменить приложение, чтобы использовать свой класс `BankStatementCSVParser`, а точнее, его метод `parseLinesFromCSV()`, как показано в примере 2-5.

Пример 2-5. Использование CSV-парсера

```
final BankStatementCSVParser bankStatementParser = new BankTransactionCSVParser();

final String fileName = args[0];
```

```
final Path path = Paths.get(RESOURCES + fileName);
final List<String> lines = Files.readAllLines(path);

final List<BankTransaction> bankTransactions
    = bankStatementParser.parseLinesFromCSV(lines);

System.out.println("The total for all transactions is " + calculateTotalAmount(bank
Transactions));
System.out.println("Transactions in January " + selectInMonth(BankTransactions,
Month.JANUARY));
```

Таким образом, вам больше не нужно знать для решения различных задач, как именно работает функция парсинга. Достаточно просто использовать объекты BankTransaction, чтобы получить требуемую информацию. В примере 2-6 показано, как объявить методы `calculateTotalAmount()` и `selectInMonth()`, предназначенные для обработки списка транзакций и выдачи соответствующего результата. В главе 3 вы познакомитесь с лямбда-выражениями и Streams API, которые помогут вам в дальнейшем сделать ваш код более аккуратным.

Пример 2-6. Обработка списков транзакций

```
public static double calculateTotalAmount(final List<BankTransaction>
bankTransactions) {
    double total = 0d;
    for(final BankTransaction bankTransaction: bankTransactions) {
        total += bankTransaction.getAmount();
    }
    return total;
}

public static List<BankTransaction> selectInMonth(final List<BankTransaction>
bankTransactions, final Month month) {

    final List<BankTransaction> bankTransactionsInMonth = new ArrayList<>();
    for(final BankTransaction bankTransaction: bankTransactions) {
        if(bankTransaction.getDate().getMonth() == month) {
            bankTransactionsInMonth.add(bankTransaction);
        }
    }
    return bankTransactionsInMonth;
}
```

Главное преимущество данного решения заключается в том, что основное приложение больше не отвечает за функцию парсинга. Теперь эта работа передана отдельному классу, а его методы могут редактироваться или изменяться независимо. По мере поступления новых требований к приложению вы можете изменять функционал, реализуемый в классе `BankStatementCSVParser`.

Кроме того, если вам нужно, скажем, изменить алгоритм парсинга (например, реализовать более сложную версию, которая кэширует результаты), то теперь достаточно внести изменения только в одном месте. Более того, вы ввели класс `BankTransaction`, благодаря которому ваше приложение может работать независимо от конкретного формата данных.

Мы считаем хорошей привычкой при реализации методов следовать «*принципу наименьшего удивления*», то есть добиваться того, чтобы назначение методов при просмотре кода было очевидным. Вот что это значит:

- Используйте логичные названия методов, отражающие их «роли» (к примеру, `calculateTotalAmount()`).
- Не изменяйте свойства параметров, так как другие части кода могут зависеть от этого.

Конечно, описанный принцип является довольно субъективным. Только вам и вашей команде решать — взять его за правило или нет.

Связность

Итак, вы познакомились с тремя принципами: KISS, DRY и SRP. Но мы все еще не рассмотрели параметры, позволяющие оценить качество вашего кода. В среде разработчиков вы будете часто слышать о *связности* как о важном качестве различных частей кода. Звучит немного замысловато, однако это действительно очень полезная концепция, благодаря которой можно легко оценить, насколько ваш код удобен в обслуживании.

Связность характеризует то, *как взаимодействуют* друг с другом различные детали. Если говорить точнее, связность — это мера того, насколько сильно взаимосвязаны задачи элементов внутри класса или метода. Другими словами, насколько узлы класса «пересекаются друг с другом». Таким способом вы можете оценить сложность своего приложения. Вам следует стремиться к достижению *высокой связности*, что подразумевает создание

легко читаемого и легко понимаемого другими кода. В коде, который мы переработали выше, класс `BankTransactionCSVParser` обладает высокой связностью. По сути, он объединяет в себе два метода, предназначенные для обработки CSV-данных.

В целом принцип связности применим к классам (внутриклассовая связность), но точно также его можно применить и к методам.

Если вы посмотрите на точку входа в нашу программу — класс `BankStatementAnalyzer` — то обнаружите, что его задача заключается только в том, чтобы связать между собой различные части программы, такие как парсинг, расчет суммы и выведение результата на экран. При этом код, который вычисляет сумму, по-прежнему находится в классе `BankStatementAnalyzer` и является его статическим методом. Это пример низкой, плохой связности, так как обязанности по вычислению суммы находятся в классе, который непосредственно для этого не предназначен.

Таким образом, лучше переместить операции вычисления в отдельный класс и назвать его `BankStatementProcessor`. Вы видите, что аргумент метода списка транзакций доступен для всех операций, так что вы можете добавить его в класс в качестве поля. В результате сигнатуры метода становятся проще, а класс `BankStatementProcessor` становится более связным. В примере 2-7 показан конечный результат. Дополнительный плюс всего этого заключается в том, что методы класса `BankStatementProcessor` могут быть использованы другими частями приложения независимо.

Пример 2-7. Группировка операций вычисления в класс `BankStatementProcessor`

```
public class BankStatementProcessor {

    private final List<BankTransaction> bankTransactions;

    public BankStatementProcessor(final List<BankTransaction> bankTransactions) {
        this.bankTransactions = bankTransactions;
    }

    public double calculateTotalAmount() {
        double total = 0;
        for(final BankTransaction bankTransaction: bankTransactions) {
            total += bankTransaction.getAmount();
        }
        return total;
```

```
    }

    public double calculateTotalInMonth(final Month month) {
        double total = 0;
        for(final BankTransaction bankTransaction: bankTransactions) {
            if(bankTransaction.getDate().getMonth() == month) {
                total += bankTransaction.getAmount();
            }
        }
        return total;
    }

    public double calculateTotalForCategory(final String category) {
        double total = 0;
        for(final BankTransaction bankTransaction: bankTransactions) {
            if(bankTransaction.getDescription().equals(category)) {
                total += bankTransaction.getAmount();
            }
        }
        return total;
    }
}
```

Теперь методы этого класса можно использовать в BankStatementAnalyzer, как показано в примере 2-8.

Пример 2-8. Обработка списка транзакций при помощи класса BankStatementProcessor

```
public class BankStatementAnalyzer {
    private static final String RESOURCES = "src/main/resources/";
    private static final BankStatementCSVParser bankStatementParser = new
BankStatementCSVParser();

    public static void main(final String... args) throws IOException {

        final String fileName = args[0];
        final Path path = Paths.get(RESOURCES + fileName);
        final List<String> lines = Files.readAllLines(path);

        final List<BankTransaction> bankTransactions = bankStatementParser.
parseLinesFrom(lines);
```

```
        final BankStatementProcessor bankStatementProcessor = new BankStatementProc
essor(bankTransactions);

        collectSummary(bankStatementProcessor);
    }

    private static void collectSummary(final BankStatementProcessor
bankStatementProcessor) {
        System.out.println("The total for all transactions is "
                + bankStatementProcessor.calculateTotalAmount());

        System.out.println("The total for transactions in January is "
                + bankStatementProcessor.calculateTotalInMonth(Month.JANUARY));

        System.out.println("The total for transactions in February is "
                + bankStatementProcessor.calculateTotalInMonth(Month.FEBRUARY));

        System.out.println("The total salary received is "
                + bankStatementProcessor.calculateTotalForCategory("Salary"));
    }
}
```

В следующих подразделах мы сфокусируем внимание на некоторых принципах, позволяющих создавать более удобный для чтения и обслуживания код.

Внутриклассовая связность

На практике вы сталкнетесь как минимум с шестью основными типами связности:

- функциональная;
- информационная;
- служебная;
- логическая;
- последовательная;
- временная.

Запомните: если методы, которые вы группируете, слабо связаны между собой, вы получите низкую связность. Сейчас мы последовательно рассмотрим

все перечисленные выше типы, после чего закрепим полученные знания при помощи тезисов, представленных в таблице 2-1.

Таблица 2-1. Плюсы и минусы различных видов связности

Тип связности	Плюсы	Минусы
Функциональная (высокая связность)	Легко читается	Может привести к избытку чересчур простых классов
Информационная (средняя связность)	Легко обслуживается	Может привести к ненужным зависимостям
Последовательная (средняя связность)	Легко обнаружить связанные операции	Не соблюдает принцип SRP
Логическая (средняя связность)	Обеспечивает определенный тип сильной категоризации	Не соблюдает принцип SRP
Служебная (слабая связность)	Легко организовать	Тяжело понять назначение класса
Временная (слабая связность)	Нет	Тяжело различить и использовать отдельные операции

Функциональная

Подход, примененный нами при разработке `BankStatementCSVParser`, состоял в функциональной группировке методов. Методы `parseFrom()` и `parseLinesFrom()` решают определенную задачу: парсинг данных из CSV-файла. По сути, метод `parseLinesFrom()` использует метод `parseFrom()`. Это хороший способ достичь высокой связности, потому что методы работают вместе. Поэтому мы их объединили, чтобы их было легче найти и понять. Опасность функциональной связности заключается в том, что можно поддаться искушению и создать чрезмерное количество слишком простых классов, имеющих по одному методу. Движение по этому пути приводит к ненужному многословию и усложняет код, так как приходится работать с необоснованно большим количеством классов.

Информационная

Другая причина для группировки методов: они работают с одними и теми же данными или доменным объектом. Скажем, вам нужно создать, считать, обновить или удалить объект `BankTransaction`. Возможно, вы решите создать класс для выполнения этих операций. В примере 2-9 показан класс, реализующий информационную связность различных методов. Каждый метод генерирует исключение `UnsupportedOperationException`, которое предупреждает о том, что тело функции не реализовано (просто для примера).

Пример 2-9. Пример информационной связности

```
public class BankTransactionDAO {

    public BankTransaction create(final LocalDate date, final double amount, final
String description) {
        // ...
        throw new UnsupportedOperationException();
    }

    public BankTransaction read(final long id) {
        // ...
        throw new UnsupportedOperationException();
    }

    public BankTransaction update(final long id) {
        // ...
        throw new UnsupportedOperationException();
    }

    public void delete(final BankTransaction BankTransaction) {
        // ...
        throw new UnsupportedOperationException();
    }
}
```

Это типовой пример, часто встречающийся при организации работы с базой данных, обслуживающей таблицу определенного доменного объекта. Часто этот шаблон называют *DAO* (*Data Access Object — объект доступа к данным*). Он требует наличия своего рода ID-номера для идентификации объекта. Обычно, DAO представляет собой абстрактный класс и инкапсулирует доступ к источнику данных, например, к обычной или резидентной базе данных.

Недостатком информационного типа связанности можно считать то обстоятельство, что при его применении группируются функции, реализующие разные задачи. Это приводит к возникновению дополнительных зависимостей в классе.

Служебная

Возможно, у вас возникнет искушение объединить несколько несвязанных методов в класс. Так случается, когда сложно определить, к чему принадлежит

метод. Тогда вы создаете некий служебный класс, который становится этаким «мастером на все руки».

На самом деле этого стоит избегать, потому что такая методика приводит к слабой связности. Если методы по смыслу не связаны друг с другом, класс в целом становится малопонятным. Кроме того, служебные классы становятся «сложно обнаруживаемыми». Вы хотите, чтобы ваш код было легко найти и легко понять. Служебные классы идут против этого принципа, потому что состоят из разных, логически не связанных друг с другом методов без четкого разделения по категориям.

Логическая

Допустим, вам нужно реализовать поддержку таких форматов как CSV, JSON и XML. Возможно, вы захотите объединить методы, предназначенные для обработки разных форматов, в один класс, как показано в примере 2-10.

Пример 2-10. Пример логической связности

```
public class BankTransactionParser {

    public BankTransaction parseFromCSV(final String line) {
        // ...
        throw new UnsupportedOperationException();
    }

    public BankTransaction parseFromJSON(final String line) {
        // ...
        throw new UnsupportedOperationException();
    }

    public BankTransaction parseFromXML(final String line) {
        // ...
        throw new UnsupportedOperationException();
    }
}
```

На самом деле данные методы логически относятся к категории «парсинга». При этом по своей сути они разные и мало связаны друг с другом. Их группировка также противоречит принципу SRP, о котором мы говорили ранее, потому что класс берет на себя ответственность за несколько различных задач. Из этого следует, что данный подход не является желательным.

Из раздела «Связанность» вы узнаете, что существуют способы решения проблемы реализации различных способов парсинга, сохранив при этом высокую связность.

Последовательная

Положим, вы хотите прочитать файл, сделать парсинг, обработать и сохранить информацию. Все эти методы можно поместить в один класс. Выходные данные чтения файла становятся входными данными для парсинга, выходные данные парсинга становятся входными данными для обработки и т. д.

Это называется последовательной связностью, так как методы группируются в класс по принципу движения потока данных, что позволяет легко понимать, как работают все операции вместе. К сожалению, на практике это означает, что у класса появляется много причин для его изменения (много разных методов), поэтому опять же нарушается принцип SRP. Кроме того, следует отметить, что могут применяться различные способы обработки данных, суммирования, сохранения и т. д., поэтому последовательная связность быстро приводит к образованию сложных классов.

Наилучший подход — реализовать каждую задачу в индивидуальном классе с хорошей связностью.

Временная

Класс с временной связностью — это класс, который выполняет несколько операций, связанных между собой только во времени. Стандартным примером такого класса является класс, который выполняет какие-либо инициализационные действия (например, подключение и отключение соединения с базой данных), которые вызываются до или после основных операций. Инициализация и другие операции не связаны между собой, но должны вызываться в строго определенное время.

Связность методов

Все принципы связности, описанные выше, можно применить и к методам. Чем сложнее функционал метода, тем сложнее понять, что же он все-таки делает. Другими словами, ваш метод обладает слабой связностью, если он обрабатывает разнородные несвязанные задачи. Методы со слабой связностью тяжелее тестировать, потому что они реализуют несколько задач, что

усложняет проверку отдельных опций. Если вы сталкиваетесь с методом, содержащим ряд блоков `if/else`, которые вносят изменения в большое количество полей класса или параметров метода, это говорит о том, что вам стоит разбить его на части с большей связностью.

Связанность

Другая важная характеристика кода — это *связанность*. Если *связность* говорит о том, насколько связаны элементы в классе, пакете или методе, то *связанность* показывает уровень зависимости от других классов. Иначе говоря, связанность показывает, сколько знаний (то есть специфических функций) вы заложили в определенный класс. Это важно, поскольку, чем больше функций возложено на класс, тем сложнее вносить в него изменения. По сути, изменения в классе влекут за собой изменения в связанных с ним классах.

Чтобы понять, что такое связанность, представьте часы. Для того чтобы посмотреть, сколько сейчас времени, не обязательно знать, как они работают. Вы не зависите от того, какая «начинка» у них внутри. То есть вы можете изменить или заменить полностью внутренности часов, но это не повлияет на считывание показаний с них. Две эти части — интерфейс и реализация — отвязаны друг от друга.

Связанность показывает, *насколько* части *зависят* друг от друга. К примеру, класс `BankStatementAnalyzer` связан с классом `BankStatementCSVParser`. Что, если вам нужно изменить парсер таким образом, чтобы он стал понимать формат JSON? А если понадобится поддержка XML? Не очень приятные изменения, верно? Но не переживайте! Вы можете разделить различные составляющие при помощи интерфейса, оптимального инструмента, обеспечивающего определенную гибкость в условиях изменения требований.

Во-первых, вам нужно ввести интерфейс, который расскажет, как использовать парсер для банковских операций, но без написания конкретной реализации, как показано в примере 2-11.

Пример 2-11. Добавление интерфейса для парсинга банковских операций

```
public interface BankStatementParser {
    BankTransaction parseFrom(String line);
    List<BankTransaction> parseLinesFrom(List<String> lines);
}
```

Теперь класс BankStatementCSVParser будет реализовывать этот интерфейс:

```
public class BankStatementCSVParser implements BankStatementParser {
    // ...
}
```

Все это хорошо, но как отвязать BankStatementAnalyzer от специфической реализации BankStatementCSVParser? Нужно использовать интерфейс! За счет введения нового метода analyze(), который принимает BankTransactionParser в качестве аргумента, вы больше не связаны с этой специфической реализацией (смотрите пример 2-12).

Пример 2-12. Разделение парсера и BankStatementAnalyzer

```
public class BankStatementAnalyzer {
    private static final String RESOURCES = "src/main/resources/";

    public void analyze(final String fileName, final BankStatementParser
bankStatementParser)
    throws IOException {

        final Path path = Paths.get(RESOURCES + fileName);
        final List<String> lines = Files.readAllLines(path);

        final List<BankTransaction> bankTransactions = bankStatementParser.
parseLinesFrom(lines);

        final BankStatementProcessor bankStatementProcessor = new BankStatementProc
essor(bankTransactions);

        collectSummary(bankStatementProcessor);
    }

    // ...
}
```

Великолепно! Теперь классу BankStatementAnalyzer больше не нужно знать, как реализуются те или иные конкретные функции, а это упрощает внесение изменений в программу.

На рис. 2-1 показана разница в зависимостях до и после разделения классов.

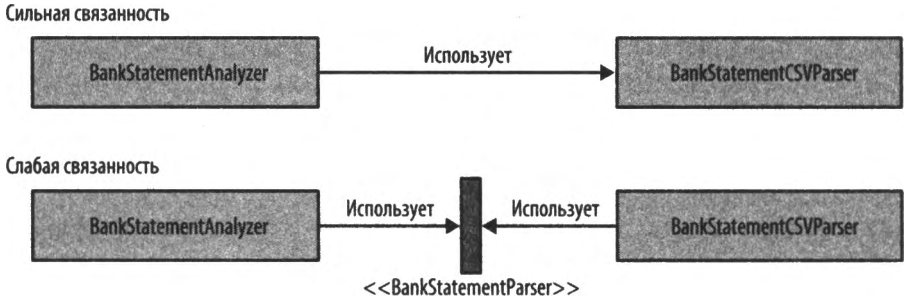

Рис. 2-1. Развязка двух классов

Теперь можно собрать все вместе и создать главный класс (пример 2-13).

Пример 2-13. Главный класс приложения

```
public class MainApplication {

    public static void main(final String... args) throws IOException {

        final BankStatementAnalyzer bankStatementAnalyzer
            = new BankStatementAnalyzer();

        final BankStatementParser bankStatementParser
            = new BankStatementCSVParser();

        bankStatementAnalyzer.analyze(args[0], bankStatementParser);
    }
}
```

В общем, при написании кода вы должны целиться на *слабую связанность*. В таком случае различные компоненты кода не зависят от внутренней реализации других компонентов. Противоположность слабой связанности — *сильная связанность*. Это то, чего следует избегать.

Тестирование

Допустим, вы написали свою программу и даже после нескольких запусков создается впечатление, что она хорошо работает. Однако насколько вы уверены, что ваш код работоспособен в любых ситуациях? Можете ли вы

гарантировать клиенту, что программа соответствует всем заявленным требованиям? В этом разделе вы познакомитесь с тестированием и узнаете, как создать свой первый автоматический тест, используя самую популярную адаптированную под Java платформу для тестирования: JUnit.

Автоматизированное тестирование

Автоматизированное тестирование звучит как еще одно занятие, которое может отнять большой кусок времени от самой веселой части — непосредственно написания кода! Почему это должно волновать вас?

Как ни странно, в программировании ничего не работает с первого раза. Поэтому очевидно, что у тестирования есть плюсы. Можете себе представить внедрение нового программного обеспечения для автопилота самолетов без тестирования?

При этом тестирование не должно быть ручным. В автоматическом тестировании есть некоторый набор тестов, которые выполняются без вмешательства человека. Если вы хотите быть уверены в том, что ваша программа ведет себя корректно и работает стабильно, тестирование должно производиться сразу после внесения изменений в программное обеспечение. В среднем профессиональный разработчик запускает сотни или даже тысячи автоматических тестов в день.

В этом разделе мы впервые кратко рассмотрим преимущества автоматизированного тестирования, чтобы вы могли четко понять, почему оно является основной частью разработки программного обеспечения.

Доверие

Во-первых, выполнение тестов для проверки поведения программы на соответствие спецификации дает вам уверенность, что вы выполнили все требования заказчика. Результаты теста вы можете предоставить клиенту как гарантию работоспособности программного продукта. По сути, тесты рождаются из спецификации (технического задания), полученной от клиента.

Устойчивость к изменениям

Во-вторых, как вы можете быть уверены, внося изменения в программу, что случайно не повредили что-нибудь? Если программа небольшая, может

показаться, что проблемы от вас не ускользнут. А что, если вы работаете с базой кода с миллионами строк? Или вносите изменения в код, который писал ваш коллега? Насколько вы будете уверены в его работоспособности? Если у вас есть набор автоматизированных тестов — очень удобно использовать их, чтобы убедиться, что вы не создали новые баги.

Понимание программы

В-третьих, автоматизированные тесты помогут вам понять, как взаимодействуют между собой различные участки исходного кода. По сути, тесты могут раскрыть зависимости различных компонентов программы и их влияние друг на друга. Согласитесь, в некоторых ситуациях получить краткое общее представление о работе программы может быть чрезвычайно полезно. Допустим, вы присоединились к новому проекту. С чего начать, где получить краткий обзор компонентов программы? Тесты — отличный способ начать.

Использование JUnit

Надеемся, вы убедились в том, что автоматизированные тесты очень важны. В этом разделе вы создадите свой первый автоматизированный тест при помощи популярного Java-фреймворка под названием *JUnit*. У всего есть своя цена. Создание тестов, в первую очередь, требует времени. Кроме того, вам придется задуматься о долгосрочном обслуживании написанных вами тестов, потому что тесты — это тоже код. Тем не менее положительные стороны тестирования, перечисленные выше, перевешивают отрицательные. В частности, вы научитесь писать *модульные тесты*, которые проверяют на корректность маленькие изолированные кусочки программы, такие как методы или маленькие классы. На протяжении всей книги вы будете получать рекомендации по созданию хороших тестов. В данной главе впервые пройдет вводный инструктаж по написанию простого теста для класса `BankTransactionCSVParser`.

Объявление метода теста

Первый вопрос, который может у вас возникнуть, — это где писать тест. Стандартное соглашение, принятое в сборщиках Maven и Gradle, призывает хранить код в *src/main/java*, а классы тестов в *src/test/java*. Вам также понадобится добавить зависимость к библиотеке JUnit в ваш проект. О том, как структурировать проекты в Maven и Gradle, вы узнаете в главе 3.

В примере 2-14 показан простой тест для класса `BankTransactionCSVParser`.

Наш тестовый класс `BankStatementCSVParserTest` имеет в названии приставку `Test`. Это не острая необходимость, но часто используется для лучшего запоминания.

Пример 2-14. Модульный тест для CSV-парсера

```
import org.junit.Assert;
import org.junit.Test;
public class BankStatementCSVParserTest {

    private final BankStatementParser statementParser = new
BankStatementCSVParser();

    @Test
    public void shouldParseOneCorrectLine() throws Exception {
        Assert.fail("Not yet implemented");
    }
}
```

В данном коде много нового. Давайте разбираться.

- Класс модульного теста — это обычный класс, который называется `BankStatementCSVParserTest`. Использование приставки `Test` в конце — общепринятая норма.
- В классе объявлен один метод: `shouldParseOneCorrectLine()`. Рекомендуется всегда давать методу название, позволяющее понять, что именно тестируется.
- Метод выделен при помощи аннотации `@Test` из JUnit. Данный знак означает, что метод представляет собой модульный тест, который должен быть выполнен. Вы можете объявить приватные вспомогательные методы в тестовом классе, но они не будут запущены при запуске теста.
- Реализация метода вызывает `Assert.fail("Not yet implemented")`, что приводит к провалу модульного теста и выдаче сообщения `Not yet implemented` («Еще не реализовано»). Скоро вы узнаете, как действительно реализовать модульный тест с использованием набора операций контроля, доступных в JUnit.

Вы можете выполнить приведенный тест в своем любимом сборщике (например, Maven или Gradle) или с помощью IDE. Например, после запуска теста в IntelliJ IDE вы получите окно, показанное на рис. 2-2. Как вы видите, тест закончился неудачно, появилось сообщение `Not yet implemented`. Теперь давайте посмотрим, как реализовать настоящий тест и убедиться в работоспособности класса `BankStatementCSVParser`.

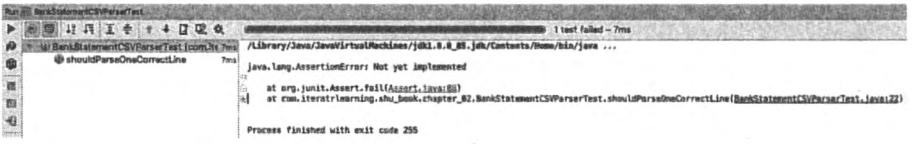

Рис. 2-2. Скриншот из среды IntelliJ IDE с результатом запуска теста

Операторы контроля

Только что вы познакомились с методом `Assert.fail()`. Это статический метод, предоставляемый в JUnit, и относится он к *операторам контроля*. JUnit поддерживает много операторов контроля, предназначенных для тестирования различных состояний. Они позволяют задать ожидаемый результат и сравнить его с фактическим результатом работы.

Один из статических методов — `Assert.assertEquals()`. Его можно использовать для проверки корректности работы метода `parseFrom()` с определенными входными данными. Рассмотрим пример 2-15.

Пример 2-15. Применение операторов контроля

```
@Test
public void shouldParseOneCorrectLine() throws Exception {
    final String line = "30-01-2017,-50,Tesco";

    final BankTransaction result = statementParser.parseFrom(line);

    final BankTransaction expected
        = new BankTransaction(LocalDate.of(2017, Month.JANUARY, 30), -50, "Tesco");
    final double tolerance = 0.0d;

    Assert.assertEquals(expected.getDate(), result.getDate());
    Assert.assertEquals(expected.getAmount(), result.getAmount(), tolerance);
    Assert.assertEquals(expected.getDescription(), result.getDescription());
}
```

Что происходит? Здесь можно выделить три составляющих.

1. Вы задаете содержимое для теста. В данном случае это строка данных.
2. Вы производите некоторое действие. В данном случае — парсинг входной строки.
3. Вы задаете операторы контроля для ожидаемых результатов. Конкретно в этом примере вы проверяете, что дата, сумма и описание считываются корректно.

Данный трехступенчатый шаблон модульного тестирования часто описывают формулой *Given-When-Then* [«Дано-Если-Тогда»]. Это довольно хороший подход, он позволяет разделить тест на этапы и понять принцип его работы.

Если вы запустите тест снова, то с определенной вероятностью увидите зеленую строку, показывающую, что тест прошел удачно (рис. 2-3).

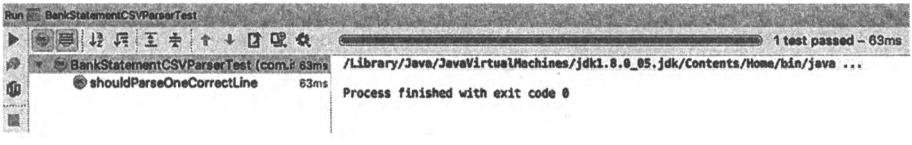

Рис. 2-3. Запуск модульного теста

Есть и другие виды операторов контроля. Они сведены в таблицу 2-2.

Таблица 2-2. Операторы контроля

Оператор контроля	Назначение
Assert.fail(message)	«Провальный тест». Его удобно использовать в качестве «заглушки», пока вы не напишете код для теста
Assert.assertEquals (expected, actual)	Тест, который проверяет эквивалентность двух значений
Assert.assertEquals (expected, actual, delta)	Тест, который проверяет два значения типа float или double на равенство с определенным допуском
Assert.assertNotNull(object)	Проверяет, что объект не null

Покрытие кода

Вы создали свой первый тест, поздравляем! Но как вы можете определить, достаточно ли этого? Покрытие кода — это показатель, описывающий,

какой процент вашего кода (какое количество строк или блоков) подвергается тестированию вашим набором тестов. Вообще, стоит нацеливаться на наибольшее покрытие, потому что в таком случае значительно снижается вероятность возникновения багов. Пока нет какого-то конкретного значения покрытия, которое считалось бы достаточным, но мы рекомендуем удерживать его в пределах не ниже 70–90%. На практике очень сложно достичь 100% покрытия кода, потому что вам придется, к примеру, тестировать геттеры и сеттеры, а они дают малый объем.

Стоит отметить, что покрытие кода — это не обязательно хороший измеритель качества тестирования кода. По сути, покрытие кода только показывает, какие участки вы еще не протестировали, но оно ничего не говорит о качестве самих тестов. Вы можете предусмотреть тесты для простейших случаев, упустив более сложные и редкие варианты отказов.

Популярными инструментами для оценки покрытия кода в Java являются *JaCoCo*, *Emma* и *Cobertura*. На практике вы будете сталкиваться с людьми, рассказывающими о линейном покрытии, которое показывает, какое количество выражений в коде покрыто тестами. Данный подход дает ложное представление о хорошем покрытии, поскольку условные выражения (`if`, `while`, `for`) в таком случае считываются как обычные выражения. При этом условные операторы могут иметь несколько возможных ветвей. Поэтому вы должны предусматривать охват всех ветвей, учитывая истинные или ложные состояния для каждой из них.

Выводы

- Огромные классы и дублирование кода ведут к созданию совершенно нечитаемого и неудобного в обслуживании кода.
- Принцип единственной ответственности помогает в создании легко управляемого и легко обслуживаемого кода.
- Связность характеризует, насколько связаны между собой элементы внутри класса или метода.
- Связанность показывает, насколько данный класс зависит от других элементов или участков кода.
- Высокая связность и слабая связанность — это признаки удобного в обслуживании кода.

- Набор автоматизированных тестов повышает вашу уверенность в корректности программного обеспечения, делает его более устойчивым к изменениям и облегчает понимание программы.
- JUnit — это Java-фреймворк для тестов, который позволяет создавать модульные тесты для проверки поведения методов и классов.
- Given-When-Then — это шаблон создания тестов в три этапа, позволяющий хорошо понимать саму реализацию теста.

Самостоятельная работа

Если вы хотите расширить и укрепить знания, полученные в этой главе, можете попробовать сделать что-нибудь из следующего:

- Напишите еще парочку модульных тестов для тестирования CSV-парсера.
- Реализуйте поддержку таких операций, как поиск самых наиболее и наименее затратных транзакций за определенный временной период.
- Сделайте возможность вывода гистограммы расходов, группируя их по месяцам и по описанию.

В завершение

Марк Эрбергцук очень счастлив и благодарен вам за реализацию Анализатора банковских операций. Он дал ему имя «THE Bank Statements Analyzer». Марку так понравилось ваше приложение, что он хочет развивать его и дальше. Он просит расширить возможности чтения, парсинга, обработки и подсчета. К примеру, он фанат формата JSON. В завершение он отметил, что тестирование сейчас несколько ограниченно и, кроме того, он даже нашел пару багов.

Вот то, чем вы будете заниматься в следующей главе, в которой вы познакомитесь с обработкой исключений, принципом открытости/закрытости и узнаете о том, как собрать Java-проект, используя инструменты сборки.

Расширяем анализатор банковских операций

Задача

Марк Эрбергцук был очень доволен работой, которую вы проделали в предыдущей главе. Вы создали базовый анализатор банковских операций, причем он оказался более-менее жизнеспособным. Поэтому Марк считает, что данный проект стоит развивать, и просит вас разработать новую версию, обладающую дополнительными возможностями.

Цель

Из предыдущей главы вы узнали, как создать приложение для анализа банковских операций, работающее с файлами в формате CSV. В процессе работы над ним вы познакомились с базовыми принципами разработки программного обеспечения, позволяющими писать обслуживаемый код в соответствии с принципом единственной ответственности, научились избегать антишаблонов (таких как «класс-бог» и дублирование кода). По мере совершенствования кода вы также познакомились с такими понятиями, как связанность (демонстрирующим зависимость от других классов) и связность (показывающим взаимосвязь между элементами класса или метода).

Тем не менее функционал приложения все еще довольно ограничен. Что, если нам реализовать возможность поиска транзакций, поддержку различных форматов, экспорт отчетов в другие форматы, такие как текст или HTML?

В этой главе вы займетесь разработкой программного обеспечения более углубленно. В первую очередь вы познакомитесь с принципом открытости/закрытости, который поможет вам сделать код более гибким и удобным в обслуживании. Вы получите ряд рекомендаций по применению интерфейсов, а также узнаете, как избежать возникновения сильной связанности. Кроме того, мы поговорим об исключениях в Java, обсудим, когда имеет смысл включать их в API, а когда нет. И наконец, вы узнаете о процессе сборки Java-приложений с использованием таких сборщиков как Maven и Gradle.

Если в какой-либо момент вам захочется взглянуть на исходный код программы для этой главы, вы можете скачать пакет в репозитории: `/com/iteratrlearning/shu_book/chapter_03`.

Требования к расширенному анализатору банковских операций

У вас состоялась дружеская переписка с Марком Эрбергцуком, благодаря которой вы собрали для себя все новые требования ко второй версии анализатора банковских выражений. Он хотел бы расширить функционал некоторых операций, поскольку на данный момент приложение довольно ограничено в своих возможностях. Сейчас оно может только подсчитывать суммы за определенный месяц или в определенных категориях. Марк просит добавить следующие опции.

1. Он хочет иметь возможность делать поиск по определенным транзакциям, например выводить список всех транзакций за конкретный период времени или транзакций в определенной категории.
2. Марк также хочет иметь возможность экспортировать статистику по результатам поиска в другой формат, такой как текст или HTML.

Данная глава будет посвящена работе над реализацией этих требований.

Принцип открытости/закрытости

Давайте начнем с простого и реализуем метод, возвращающий все транзакции с суммой, превышающей заданную. Первый вопрос: в каком месте объявить этот метод? Можно создать отдельный класс `BankTransactionFinder`, в котором

будет находиться простой метод `findTransactions()`. Однако напомним, что в предыдущей главе мы создали класс `BankTransactionProcessor`. Так что же делать? В данной ситуации не так много плюсов от создания нового класса, ведь вам нужен всего один новый метод. Новый класс сделает проект сложнее, так как поспособствует увеличению количества названий, что, в свою очередь, затруднит понимание зависимостей между различными объектами. Объявление метода в классе `BankTransactionProcessor` улучшит читаемость кода. Вы сразу будете понимать, что в данном классе собраны все методы, которые выполняют разного рода обработку данных. После того как мы определились, где поместить метод, можно заняться его реализацией (пример 3-1).

Пример 3-1. Поиск транзакций на сумму больше заданной

```
public List<BankTransaction> findTransactionsGreaterThanEqual(final int amount) {
    final List<BankTransaction> result = new ArrayList<>();
    for(final BankTransaction bankTransaction: bankTransactions) {
        if(bankTransaction.getAmount() >= amount) {
            result.add(bankTransaction);
        }
    }
    return result;
}
```

Здесь все достаточно рационально. Но что, если вы хотите сделать поиск в конкретном месяце? Тогда придется дублировать данный метод, как показано в примере 3-2.

Пример 3-2. Поиск транзакций в определенном месяце

```
public List<BankTransaction> findTransactionsInMonth(final Month month) {
    final List<BankTransaction> result = new ArrayList<>();
    for(final BankTransaction bankTransaction: bankTransactions) {
        if(bankTransaction.getDate().getMonth() == month) {
            result.add(bankTransaction);
        }
    }
    return result;
}
```

В предыдущей главе мы познакомились с понятием дублирования кода. Наш код уже начинает «пахнуть» повышенной хрупкостью, особенно если

требования к приложению будут часто меняться. Например, если в будущем вам понадобится изменить логику перебора позиций, то вам придется вносить эти изменения несколько раз.

Данный подход не будет работать и с более сложными требованиями. Что если нам понадобится искать транзакции не только на определенную сумму, но одновременно и за конкретный месяц? Реализовать это новое требование можно так, как показано в примере 3-3.

Пример 3-3. Поиск банковских операций на определенную сумму и за определенный месяц

```
public List<BankTransaction> findTransactionsInMonthAndGreater(final Month month,
final int amount) {
    final List<BankTransaction> result = new ArrayList<>();
    for(final BankTransaction bankTransaction: bankTransactions) {
        if(bankTransaction.getDate().getMonth() == month && bankTransaction.
getAmount() >= amount) {
            result.add(bankTransaction);
        }
    }
    return result;
}
```

По правде говоря, такой подход имеет ряд недостатков:

- Код становится более сложным, поскольку вам нужно объединить несколько свойств банковских операций.
- Алгоритм выбора подходящих транзакций объединен с алгоритмом их последовательного перебора, что усложняет их разделение.
- Вы продолжаете заниматься дублированием.

Вот где на сцену выходит принцип открытости/закрытости. Он продвигает идею возможности изменения поведения метода или класса без необходимости изменения самого кода. То есть, если говорить о нашем примере, нужно расширить возможности метода `findTransactions()` без дублирования или изменения кода. Разве это реально? Как мы говорили выше, концепции последовательного перебора и логика выбора подходящих транзакций связаны между собой. В предыдущей главе вы познакомились с отличным инструментом для разделения задач, а именно — с интерфейсами. Сейчас мы создадим интерфейс `BankTransactionFilter` и возложим на него

ответственность за выбор подходящих транзакций из списка. В нем будет располагаться метод `test()`, возвращающий булево значение, а в качестве аргумента принимающий объект `BankTransaction`. Таким образом метод `test()` будет иметь доступ ко всем свойствам объекта `BankTransaction`, что позволит ему установить любые критерии поиска.

Интерфейс, содержащий только один абстрактный метод, называется *функциональным интерфейсом* (начиная с Java версии 8). Вы можете аннотировать данный интерфейс при помощи строки `@FunctionalInterface`, чтобы сделать его назначение более понятным.

Пример 3-4. Интерфейс `BankTransactionFilter`

```
@FunctionalInterface
public interface BankTransactionFilter {
    boolean test(BankTransaction bankTransaction);
}
```

В Java 8 появился базовый интерфейс `java.util.function.Predicate<T>`, который мог бы стать отличным решением нашей проблемы. Однако в этой главе мы уже ввели собственный интерфейс, чтобы не усложнять код слишком рано.

Интерфейс `BankTransactionFilter` представляет собой механизм выбора критериев для объектов `BankTransaction`. Теперь метод `findTransactions()` можно отредактировать так, как показано в примере 3-5. Это довольно важный момент, поскольку только что вы применили на практике новый способ разделения двух алгоритмов. Ваш метод больше не зависит от одной конкретной реализации фильтра. Вы можете придумать другие реализации, просто передавая их в качестве аргументов, без необходимости изменять сам метод. По сути, теперь метод открыт для расширения и закрыт для изменения. Благодаря этому значительно снижается вероятность возникновения новых багов, так как минимизируется количество изменений, которые нужно внести в уже отлаженный и протестированный код. Другими словами, старый код работает как и раньше и остается нетронутым.

Пример 3-5. Гибкая реализация метода `findTransactions()`, построенная на принципе открытости/закрытости

```
public List<BankTransaction> findTransactions(final BankTransactionFilter bankTran
sactionFilter) {
```

```
    final List<BankTransaction> result = new ArrayList<>();
    for(final BankTransaction bankTransaction: bankTransactions) {
        if(bankTransactionFilter.test(bankTransaction)) {
            result.add(bankTransaction);
        }
    }
    return result;
}
```

Создание экземпляра функционального интерфейса

Марк Эрбергцук счастлив. Теперь вы можете оперативно подстраивать программу под любые требования за счет метода `findTransactions()` из класса `BankTransactionProcessor`, который реализует соответствующий интерфейс `BankTransactionFilter`. Этого можно достичь, реализовав класс так, как показано в примере 3-6, а затем, передав объект в качестве аргумента методу `findTransactions()`, как показано в примере 3-7.

Пример 3-6. Объявляем класс, который реализует интерфейс `BankTransactionFilter`

```
class BankTransactionIsInFebruaryAndExpensive implements BankTransactionFilter {

    @Override
    public boolean test(final BankTransaction bankTransaction) {
        return bankTransaction.getDate().getMonth() == Month.FEBRUARY
                && bankTransaction.getAmount() >= 1_000);
    }
}
```

Пример 3-7. Вызов метода `findTransactions()` с определенной реализацией `BankTransactionFilter`

```
final List<BankTransaction> transactions
    = bankStatementProcessor.findTransactions(new BankTransactionIsInFebruaryAndEx
pensive());
```

Лямбда-выражения

Как бы там ни было, вам придется создавать новые классы каждый раз, когда будут появляться новые требования к программе. Этот процесс может

стать причиной некой шаблонности и в скором времени привести к загромождению кода. Начиная с Java версии 8 в вашем распоряжении есть *лямбда-выражения* (пример 3-8). Пока вам не стоит волноваться о синтаксисе применения этих выражений. Более подробно мы познакомимся с лямбда-выражениями и ссылками на методы в главе 7. Сейчас же вы можете представить себе это примерно так: вместо передачи объекта, который реализует какой-то интерфейс, мы передаем блок кода (по сути, функцию без названия). `bankTransaction` — это имя параметра, а стрелка -> отделяет параметр от тела лямбда-выражения, которое, по сути, представляет собой кусочек кода, проверяющий транзакцию на соответствие некоторым условиям.

Пример 3-8. Реализация `BankTransactionFilter` с использованием лямбда-выражения

```
final List<BankTransaction> transactions
    = bankStatementProcessor.findTransactions(bankTransaction ->
                bankTransaction.getDate().getMonth() == Month.FEBRUARY
                && bankTransaction.getAmount() >= 1_000);
```

В заключение скажем, что принцип открытости/закрытости очень удобен, его стоит придерживаться по нескольким причинам:

- Он снижает хрупкость кода за счет того, что уже существующий код не изменяется.
- Он поддерживает идею повторного использования существующего кода, а значит, позволяет избегать дублирования.
- Он продвигает «развязывание», а это ведет к улучшению обслуживаемости кода.

Подводные камни интерфейсов

Какое-то время назад вы познакомились с гибким методом поиска транзакций, соответствующих заданным критериям. По ходу изменения кода нашей программы возникает вопрос о том, что же будет с остальными методами, объявленными в классе `BankTransactionProcessor`. Стоит ли трансформировать их в интерфейсы? Или, может быть, переместить в другой класс? После всего у нас остаются три связанных между собой метода, которые мы создали в предыдущей главе:

- `calculateTotalAmount()`
- `calculateTotalInMonth()`
- `calculateTotalForCategory()`

Существует подход, который может несколько разочаровать вас в плане применения на практике. Он заключается в том, чтобы разместить все в одном интерфейсе. В таком случае мы получаем «интерфейс-бог».

Интерфейс-бог

Кто-то из вас, возможно, согласится с тем, что класс `BankTransactionProcessor` работает как API. Данная точка зрения может привести к тому, что вам захочется создать интерфейс, позволяющий избавиться от многочисленных реализаций обработчика банковских операций (пример 3-9). В этом интерфейсе будут содержаться все операции, которые должен выполнять обработчик банковских операций.

Пример 3-9. Интерфейс-бог

```
interface BankTransactionProcessor {
    double calculateTotalAmount();
    double calculateTotalInMonth(Month month);
    double calculateTotalInJanuary();
    double calculateAverageAmount();
    double calculateAverageAmountForCategory(Category category);
    List<BankTransaction> findTransactions(BankTransactionFilter
bankTransactionFilter);
}
```

Стоит знать и учитывать недостатки такого подхода. Во-первых, интерфейс стремительно растет и усложняется, так как каждая простая вспомогательная операция является частью большого API. Во-вторых, он начинает чем-то походить на класс-бог, с которым вы уже знакомы. По сути, интерфейс становится мешком, в который складываются всевозможные операции. Что еще хуже, вам суждено столкнуться с еще двумя формами дополнительной связанности.

- Интерфейс в Java являет собой контракт, которого должна придерживаться каждая конкретная реализация. Другими словами, каждая реализация интерфейса должна реализовывать все его функции. Это означает,

что любое изменение самого интерфейса автоматически подразумевает необходимость обновления его реализаций. Большинство новых функций и новых изменений увеличивают вероятность возникновения ошибок по цепочке.

- Конкретные свойства `BankTransaction`, такие как месяц и категория, появляются как имена методов: `calculateAverageForCategory()` и `calculateTotalInJanuary()`. Это еще одна проблема интерфейсов: теперь они зависят от определенных средств доступа доменного объекта. Если изменяется содержимое этого доменного объекта, значит, изменения грядут и в интерфейсе, и, как следствие, во всех его реализациях тоже.

Именно по этим причинам рекомендуется создавать маленькие интерфейсы. Идея в том, чтобы минимизировать зависимости от множества операций или внутренних данных доменного объекта.

Слишком мизерный

Только что мы пришли к выводу, что чем меньше интерфейс, тем лучше. Другая крайность — создавать интерфейсы для каждой отдельной операции, как показано в примере 3-10. В данном случае все интерфейсы реализует класс `BankTransactionProcessor`.

Пример 3-10. Очень маленькие интерфейсы

```
interface CalculateTotalAmount {
    double calculateTotalAmount();
}

interface CalculateAverage {
    double calculateAverage();
}

interface CalculateTotalInMonth {
    double calculateTotalInMonth(Month month);
}
```

Такой подход тоже не способствует улучшению обслуживаемости кода. И вообще, он вносит в код «антисвязность». Другими словами, становится намного сложнее найти интересующую вас операцию, так как они все

разбросаны по разным интерфейсам. Одним из признаков хорошей обслуживаемости является быстрый поиск нужных операций в коде. К тому же из-за сильной раздробленности интерфейсов код становится чрезмерно сложным, особенно в отношении отслеживания новых типов, возникающих за счет интерфейсов.

Явный API против неявного

И все-таки, каким должен быть прагматичный подход? Мы рекомендуем придерживаться принципа открытости/закрытости, чтобы сохранять гибкость операций, а наиболее популярные операции объявлять в виде элементов класса. Они могут быть реализованы более общими методами. В таком случае применение интерфейса не сильно обосновано, поскольку мы не планируем иметь различные реализации `BankTransactionProcessor`. У всех этих методов нет какой-то особой специализации, которая была бы полезна всему приложению. Поэтому нет необходимости что-то мудрить и добавлять лишнюю абстракцию в программу. `BankTransactionProcessor` — это простой класс, который обеспечивает выполнение статистических операций по транзакциям.

Отсюда возникает вопрос о том, где объявлять такие методы, как `findTransactionsGreaterThanEqual()`, при учете, что их легко можно реализовать в более общих методах вроде `findTransactions()`. Эту дилемму часто называют проблемой явного или неявного API.

На самом деле здесь есть две стороны медали. С одной стороны, метод `findTransactionsGreaterThanEqual()` является довольно очевидным и простым в использовании. Вам не придется волноваться о добавлении описания в название метода, чтобы упростить читаемость и понимание вашего API. При этом данный метод довольно ограничен в своем применении, и у вас есть все шансы получить переизбыток методов, если нужно будет реализовывать дополнительные возможности. С другой стороны, такой метод, как `findTransactions()`, изначально является более сложным в использовании и должен быть хорошо задокументирован. При этом он обеспечивает унифицированный API и участвует во всех случаях поиска транзакций. Нет четкого правила о том, что лучше, а что хуже. Все зависит от того, какие запросы вы реализуете. Если `findTransactionsGreaterThanEqual()` будет более общей (часто используемой) операцией, то имеет смысл извлечь ее в отдельный API, тем самым упростив ее применение.

Последняя реализация `BankTransactionProcessor` показана в примере 3-11.

Пример 3-11. Ключевые операции класса `BankTransactionProcessor`

```
@FunctionalInterface
public interface BankTransactionSummarizer {
    double summarize(double accumulator, BankTransaction bankTransaction);
}

@FunctionalInterface
public interface BankTransactionFilter {
    boolean test(BankTransaction bankTransaction);
}

public class BankTransactionProcessor {

    private final List<BankTransaction> bankTransactions;

    public BankStatementProcessor(final List<BankTransaction> bankTransactions) {
        this.bankTransactions = bankTransactions;
    }

    public double summarizeTransactions(final BankTransactionSummarizer
bankTransactionSummarizer) {
        double result = 0;
        for(final BankTransaction bankTransaction: bankTransactions) {
            result = bankTransactionSummarizer.summarize(result, bankTransaction);
        }
        return result;
    }

    public double calculateTotalInMonth(final Month month) {
        return summarizeTransactions((acc, bankTransaction) ->
                bankTransaction.getDate().getMonth() == month ? acc +
bankTransaction.getAmount() : acc
        );
    }
        // ...

    public List<BankTransaction> findTransactions(final BankTransactionFilter
bankTransactionFilter) {
        final List<BankTransaction> result = new ArrayList<>();
        for(final BankTransaction bankTransaction: bankTransactions) {
            if(bankTransactionFilter.test(bankTransaction)) {
                result.add(bankTransaction);
            }
        }
        return bankTransactions;
```

```
    }

    public List<BankTransaction> findTransactionsGreaterThanEqual(final int amount)
{
        return findTransactions(bankTransaction -> bankTransaction.getAmount() >=
amount);
    }

    // ...
}
```

Большинство шаблонов агрегации, с которыми вы встречались, могли быть реализованы при помощи Streams API, который появился в Java 8. Например, поиск транзакций можно легко реализовать вот так:

```
bankTransactions
    .stream()
    .filter(bankTransaction -> bankTransaction.getAmount() >= 1_000)
    .collect(toList());
```

Тем не менее Streams API построен на тех же принципах, с которыми вы только что познакомились.

Доменный класс или примитив?

Пока мы сохраняем интерфейс `BankTransactionSummarizer` простым, предпочтительно не возвращать никаких примитивов вроде `double` по результатам работы. Причина в снижении гибкости в случае возврата множественного результата. Например, метод `summarizeTransaction()` возвращает `double`. Если вы соберетесь изменить сигнатуры результата, чтобы возвращать больше значений, вам придется менять все реализации `BankTransactionProcessor`.

Решить данную проблему можно при помощи создания доменного класса под названием `Summary`, который «обернет» значение `double`. Тогда в будущем вы сможете добавлять другие поля в этот класс. Такая техника помогает «развязать» различные операции в домене и способствует минимизации изменений.

Примитив `double` имеет ограниченное число бит (разрядность), поэтому при хранении десятичных чисел мы имеем ограниченную точность. В качестве альтернативы можно рассмотреть класс `java.math.BigDecimal`, обеспечивающий произвольную точность. Однако стоит учитывать, что за эту точность придется заплатить ресурсами процессора и памяти.

Множественный экспорт

В предыдущем разделе вы познакомились с принципом открытости/закрытости и углубили свои знания в области интерфейсов Java. Все это вам очень пригодится, так как у Марка Эрбергцука появилось новое требование! Вам нужно экспортировать итоговую статистику по выбранному списку транзакций в различные форматы, включая текст, HTML, JSON и т. д. С чего начать?

Знакомство с доменным объектом

Для начала вам необходимо определиться с тем, что именно хочет экспортировать пользователь. Есть несколько возможных вариантов.

Число

Допустим, пользователю интересен только результат какой-нибудь операции вроде `calculateAverageInMonth`. Значит, в качестве результата должен выдаваться тип `double`. Хотя это и самый простой подход (как мы уже отметили ранее), он является наименее гибким и тяжелее всего поддается изменениям. Представьте, что ваш экспортер на входе принимает тип `double`. Если вдруг поменяются требования к экспорту данных, вам придется изменять код везде, где вызывается этот экспортер, что способно привести к возникновению багов.

Коллекция

Возможно, пользователь хочет экспортировать список транзакций. Например, результат работы метода `findTransaction()`. Для обеспечения гибкости в будущем, вы даже могли бы экспортировать `Iterable`. Такой подход повышает гибкость, однако в то же время он привязывает вас к необходимости экспорта именно коллекций. А что, если вам понадобится возвращать множественный результат вроде списков или другой информации?

Специализированный доменный объект

Вы можете ввести некую новую «сущность» `SummaryStatistics`, представляющую итоговую информацию, которую хотел бы видеть пользователь в результате экспорта. Доменный объект — это просто экземпляр класса, привязанный к вашему домену. Внедряя доменный объект, вы применяете одну из форм «развязывания». По сути, если возникают новые требования

к экспорту, вы можете просто добавить их в ваш новый класс. При этом вам не придется вносить кучу изменений.

Более сложный доменный объект

Еще один вариант — внедрить такой элемент как `Report`. Он является более общим и может содержать различного рода поля, которые хранят данные разных типов, включая коллекции транзакций. Нужно вам это или нет — зависит от требований пользователя и от ваших ожиданий относительно поступающей информации. Преимущество такого подхода заключается в том, что вы снова можете развязать различные части приложения: те, что производят объекты `Report`, и те, что эти объекты «потребляют».

В нашем случае предлагаем ввести доменный объект, который будет хранить итоговую статистику о списке транзакций. Его код показан в примере 3-12.

Пример 3-12. Доменный объект, хранящий статистическую информацию

```
public class SummaryStatistics {
    private final double sum;
    private final double max;
    private final double min;
    private final double average;

    public SummaryStatistics(final double sum, final double max, final double min,
final double average) {
        this.sum = sum;
        this.max = max;
        this.min = min;
        this.average = average;
    }

    public double getSum() {
        return sum;
    }

    public double getMax() {
        return max;
    }

    public double getMin() {
        return min;
    }
```

```
    public double getAverage() {
        return average;
    }
}
```

Объявление и реализация соответствующего интерфейса

Итак, вы определились с тем, какие данные нужно экспортировать. Самое время заняться API. Вам нужно создать интерфейс с именем `Exporter`. Причина, по которой мы вводим интерфейс — он дает возможность избавиться от связи с множеством реализаций экспортеров. Это как раз соответствует принципу открытости/закрытости, с которым вы познакомились ранее. На самом деле если вам понадобится заменить реализацию экспортера в JSON на экспортер в XML, вы все сделаете достаточно просто, учитывая, что они оба будут реализованы в одном интерфейсе. Первым шагом в создании такого интерфейса может стать код, приведенный в примере 3-13. Метод `export()` принимает в качестве параметра объект `SummaryStatistics`, а возвращает `void`.

Пример 3-13. Плохой интерфейс экспортера

```
public interface Exporter {
    void export(SummaryStatistics summaryStatistics);
}
```

Такого подхода следует избегать по нескольким причинам:

- Методы с возвращаемым значением типа `void` неудобны и сложны в использовании, поскольку вам неизвестно, какое именно значение возвращается. Сигнатура метода `export()` подразумевает, что где-то происходит изменение состояния чего-либо или что данный метод будет фиксировать или выводить информацию на экран. Мы не знаем!
- Возвращаемый тип `void` сильно усложняет тестирование с использованием утверждений. Мы не знаем, каков фактический результат работы метода. Что сравнивать с ожидаемым результатом?

Учитывая все вышесказанное, вам следует придумать новый API, который возвращает тип `String` (пример 3-14). Теперь понятно, что `Exporter` возвращает текстовые данные, которые затем передаются в другую часть программы для распечатки, сохранения в файл или даже отправки по электронной почте. Текстовые строки также достаточно удобны для тестирования при помощи утверждений.

Пример 3-14. Хороший интерфейс экспортера

```
public interface Exporter {
    String export(SummaryStatistics summaryStatistics);
}
```

После объявления API для экспорта информации можно реализовать различные виды экспортеров, взаимодействующих с интерфейсом `Exporter`. Вариант реализации простейшего HTML-экспортера показан в примере 3-15.

Пример 3-15. Реализация интерфейса `Exporter`

```
public class HtmlExporter implements Exporter {
    @Override
    public String export(final SummaryStatistics summaryStatistics) {

        String result = "<!doctype html>";
        result += "<html lang='en'>";
        result += "<head><title>Bank Transaction Report</title></head>";
        result += "<body>";
        result += "<ul>";
        result += "<li><strong>The sum is</strong>: " + summaryStatistics.getSum()
+ "</li>";
        result += "<li><strong>The average is</strong>: " + summaryStatistics.
getAverage() + "</li>";
        result += "<li><strong>The max is</strong>: " + summaryStatistics.getMax()
+ "</li>";
        result += "<li><strong>The min is</strong>: " + summaryStatistics.getMin()
+ "</li>";
        result += "</ul>";
        result += "</body>";
        result += "</html>";
        return result;
    }
}
```

Обработка исключений

Давненько мы не разговаривали о том, что происходит, когда что-то идет не так. Можете представить себе ситуации некорректной работы программы банковского анализатора? К примеру:

- Что, если синтаксический анализ данных проходит неправильно?
- Что, если не удается прочитать CSV-файл, содержащий информацию о банковских операциях?
- Что, если аппаратному обеспечению, на котором работает ваше приложение, не хватает ресурсов? Например оперативной памяти или места на диске?

В таких случаях вы должны получить сообщение об ошибке с информацией о трассировке стека, показывающей источник проблемы. Фрагменты в примере 3-16 показывают, как могут выглядеть такие ошибки.

Пример 3-16. Неожиданные ошибки

```
Exception in thread "main" java.lang.ArrayIndexOutOfBoundsException: 0

Exception in thread "main" java.nio.file.NoSuchFileException: src/main/resources/
bank-data-simple.csv

Exception in thread "main" java.lang.OutOfMemoryError: Java heap space
```

Для чего нужны исключения?

На некоторое время остановимся на `BankStatementCSVParser`. Как нам обрабатывать проблемы в процессе парсинга? К примеру, строка CSV-файла может быть записана не так, как вы ожидали:

- Строка CSV-файла может содержать больше, чем три ожидаемых колонки.
- Строка CSV-файла может содержать меньше, чем три ожидаемых колонки.
- Формат данных в некоторых колонках может быть некорректным (например формат даты).

Когда-то, в страшные времена программирования на языке C, вам пришлось бы добавлять очень много проверок при помощи условий `if`, которые возвращали бы коды критических ошибок. У такого подхода есть недостатки. Во-первых, он основывался на глобальных изменяемых значениях, чтобы найти самую последнюю ошибку. Это усложняло понимание отдельных частей кода. Как следствие, код становилось тяжелее обслуживать. Во-вторых, такой подход провоцировал появление ошибок, поскольку вам нужно было

различать между собой реальные значения и коды ошибок, закодированные в виде обычных значений. Система типов в этом случае была слабой и могла бы быть более полезной для программиста. И наконец, управляющий поток смешивался с бизнес-логикой, что приводило к сложному в обслуживании и тестировании коду.

Для решения упомянутых проблем в Java предусмотрен механизм исключений, имеющий множество преимуществ:

Документация

Язык поддерживает исключения как часть сигнатур методов.

Безопасность типов

Система типов определяет, обрабатываете ли вы поток исключений.

Разделение задач

Бизнес-логика и исключения отделены друг от друга при помощи блока `try\ catch`.

Проблема заключается лишь в том, что исключения, являясь особенностью языка, повышают его сложность. Вы должны знать, что в Java различают два типа исключений.

Проверяемые исключения

Это ошибки, появления которых вы ожидали и были готовы обработать. В Java вы должны объявлять метод со списком исключений, которые он может генерировать. Если вы этого не делаете, тогда вам нужно обеспечивать наличие локальных блоков `try\catch` для конкретных исключений.

Непроверяемые исключения

Это ошибки, которые могут возникнуть в любой момент выполнения программы. Методы не обязательно должны явно объявлять эти исключения в своей сигнатуре, а вызов не обязательно должен их явно обрабатывать, как это происходит с проверяемыми исключениями.

Классы Java-исключений организованы в четкой иерархии (рис. 3-1). Классы `Error` и `RuntimeException` (являются подклассами `Throwable`) — непроверяемые исключения. Не стоит ожидать, что вы сможете «выловить» их

и обработать. Класс `Exception` обычно представляет те ошибки, которые программа способна обработать.

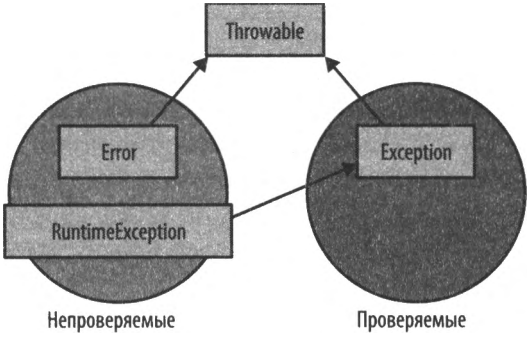

Рис. 3-1. Иерархия исключений в Java

Шаблоны и антишаблоны для исключений

Когда и какие типы исключений нужно применять? Вы удивитесь, как можно изменить BankStatementParser API, реализовав поддержку исключений. Тем не менее, к сожалению, однозначного ответа на заданный выше вопрос не существует. Потребуется немного прагматизма при принятии решения о том, какой подход является правильным для вас.

Есть две ключевых задачи при обработке CSV-файла:

- Правильно обработать синтаксис (CSV, JSON).
- Проверить корректность данных (к примеру, длина описания должна быть менее 100 символов).

Сначала рассмотрим ошибки синтаксиса, а затем перейдем к корректности данных.

Выбор между проверяемыми и непроверяемыми исключениями

Могут возникнуть ситуации, когда CSV-файл не будет соответствовать требованиям по синтаксису. Например, из-за отсутствия разделительных запятых. Игнорирование подобных проблем приводит к ошибкам во время выполнения программы. Одним из плюсов исключений в коде является более простая диагностика проблем пользователем. Соответственно, вы решаете добавить простую проверку (пример 3-17), которая генерирует исключение `CSVSyntaxException`.

Пример 3-17. Генерация исключения по синтаксису

```
final String[] columns = line.split(",");

if(columns.length < EXPECTED_ATTRIBUTES_LENGTH) {
    throw new CSVSyntaxException();
}
```

Каким должно быть исключение `CSVSyntaxException`: проверяемым или непроверяемым? Чтобы ответить на этот вопрос, нужно знать, будете ли вы требовать от пользователя вашего API совершения каких-либо действий. Например, пользователь может сделать вторую попытку в случае, если ошибка временная, или вывести сообщение на экран, что добавит изящности приложению. Обычно ошибки, связанные с основным алгоритмом программы (например, неверный формат данных или ошибка в арифметике), обрабатываются непроверяемыми исключениями, поскольку иначе они заполнили бы код большим количеством блоков `try\catch`. Кроме того, не всегда очевидно, каким должен быть правильный механизм обработки ошибки. Следовательно, нет никакого смысла навязывать его пользователю вашего API. В дополнение скажем, что системные ошибки (вроде недостаточного места на диске) тоже должны обрабатываться непроверяемыми исключениями, потому что пользователь все равно не сможет ничего сделать. В двух словах: лучше использовать непроверяемые исключения и очень умеренно проверяемые, чтобы избежать ненужного беспорядка в коде.

Теперь, когда мы убедились в целостности CSV-файла, можно приступить к решению вопроса корректности данных. Вам предстоит познакомиться с двумя распространенными антишаблонами использования исключений, после чего мы рассмотрим шаблон уведомления, предоставляющий обслуживаемое решение проблемы.

Слишком специфические

Первый вопрос, который может у вас возникнуть, — где разметить код проверки корректности данных. Вы могли бы получить его во время создания объекта `BankStatement`. Хотя, по некоторым причинам, мы рекомендуем создать специальный класс `Validator`. Вот почему:

- Вам не придется дублировать алгоритмы проверки корректности при необходимости повторного использования.

- Вы будете уверены, что различные части вашей системы выполняют проверку корректности одинаково.
- Такой алгоритм легко отдельно протестировать.
- Данный подход соответствует принципу SRP, что упрощает обслуживание и снижает сложность программы.

Существуют различные подходы к реализации такого алгоритма при помощи исключений. Один из них (слишком специфический) продемонстрирован в примере 3-18. В данном случае вы продумали все возможные непредвиденные ситуации при проверке входных данных и каждую из них преобразовали в проверяемое исключение. Исключения `DescriptionTooLongException`, `InvalidDateFormat`, `DateInTheFutureException` и `InvalidAmountException` — это объявленные пользователем проверяемые исключения (то есть они расширяют класс `Exception`). Хотя такой подход дает возможность конкретизировать механизмы обработки для каждого исключения, он абсолютно не продуктивен, поскольку требует тонкой настройки, объявляет много исключений и вынуждает пользователя индивидуально работать с каждым исключением. А это противоречит цели упростить понимание и использование API пользователем. Кроме того, вы не можете помещать все исключения в коллекцию, чтобы предоставить пользователю в виде списка.

Пример 3-18. Слишком специфические исключения

```
public class OverlySpecificBankStatementValidator {

    private String description;
    private String date;
    private String amount;

    public OverlySpecificBankStatementValidator(final String description, final
String date, final String amount) {
        this.description = Objects.requireNonNull(description);
        this.date = Objects.requireNonNull(description);
        this.amount = Objects.requireNonNull(description);
    }

    public boolean validate() throws DescriptionTooLongException,
                                     InvalidDateFormat,
                                     DateInTheFutureException,
                                     InvalidAmountException {
```

```
        if(this.description.length() > 100) {
            throw new DescriptionTooLongException();
        }

        final LocalDate parsedDate;
        try {
            parsedDate = LocalDate.parse(this.date);
        }
        catch (DateTimeParseException e) {
            throw new InvalidDateFormat();
        }
        if (parsedDate.isAfter(LocalDate.now())) throw new
DateInTheFutureException();

        try {
            Double.parseDouble(this.amount);
        }
        catch (NumberFormatException e) {
            throw new InvalidAmountException();
        }
        return true;
    }
}
```

Слишком однообразные

Еще один вариант — сделать все исключения непроверяемыми. Например, использовать `IllegalArgumentException`. В примере 3-19 показана реализация метода `validate()`, отражающая данный подход. Теперь проблема заключается в том, что вы просто не можете реализовать индивидуальную логику обработки, поскольку все исключения одинаковые. Кроме того, вы все еще не можете собрать полностью все ошибки.

Пример 3-19. Исключения `IllegalArgument` повсюду

```
public boolean validate() {

    if(this.description.length() > 100) {
        throw new IllegalArgumentException("The description is too long");
    }

    final LocalDate parsedDate;
    try {
```

```
            parsedDate = LocalDate.parse(this.date);
        }
        catch (DateTimeParseException e) {
            throw new IllegalArgumentException("Invalid format for date", e);
        }
        if (parsedDate.isAfter(LocalDate.now())) throw new
IllegalArgumentException("date cannot be in the future");

        try {
            Double.parseDouble(this.amount);
        }
        catch (NumberFormatException e) {
            throw new IllegalArgumentException("Invalid format for amount", e);
        }
        return true;
    }
```

Далее вы узнаете о шаблоне уведомления, который позволяет компенсировать недостатки слишком специфического и слишком однообразного антишаблона.

Шаблон уведомления

Шаблон уведомления преследует цель решить проблему использования слишком большого количества непроверяемых исключений.

Решение заключается в коллекционировании ошибок доменным классом[1].

Первое, что вам нужно, — это класс `Notification`, отвечающий за сбор ошибок. Объявляем его как показано в примере 3-20.

Пример 3-20. Внедрение доменного класса `Notification`, который отвечает за сбор ошибок

```
public class Notification {
    private final List<String> errors = new ArrayList<>();

    public void addError(final String message) {
        errors.add(message);
    }
```

[1] Этот шаблон впервые был предложен Мартином Фаулером. — *Прим. авт.*

```
    public boolean hasErrors() {
        return !errors.isEmpty();
    }

    public String errorMessage() {
        return errors.toString();
    }

    public List<String> getErrors() {
        return this.errors;
    }
}
```

Плюсом создания данного класса является то, что теперь вы можете объявить валидатор (метод для проверки корректности данных), который способен собирать много ошибок за раз. Это было невозможно в двух предыдущих подходах. Теперь вместо генерирования исключений вы можете просто добавлять сообщения в объект `Notification` (пример 3-21).

Пример 3-21. Шаблон уведомления

```
public Notification validate() {

    final Notification notification = new Notification();
    if(this.description.length() > 100) {
        notification.addError("The description is too long");
    }

    final LocalDate parsedDate;
    try {
        parsedDate = LocalDate.parse(this.date);
        if (parsedDate.isAfter(LocalDate.now())) {
            notification.addError("date cannot be in the future");
        }
    }
    catch (DateTimeParseException e) {
        notification.addError("Invalid format for date");
    }

    final double amount;
    try {
        amount = Double.parseDouble(this.amount);
    }
```

```
    catch (NumberFormatException e) {
        notification.addError("Invalid format for amount");
    }
    return notification;
}
```

Методика применения исключений

Теперь, когда вы узнали, в каких ситуациях следует применять исключения, давайте обсудим некоторые методические рекомендации, которые позволят вам применять исключения эффективно.

Не игнорируйте исключение

Игнорировать исключения — плохая идея. Так вы не сможете узнать, в чем на самом деле кроется проблема. Если нет очевидного механизма обработки, генерируйте непроверяемое исключение. В таком случае, если вам действительно понадобится обработать проверяемое исключение, вы будете вынуждены вернуться и разобраться с ним после того, как увидите проблему во время выполнения программы.

Не перехватывайте «общие» исключения

По возможности перехватывайте конкретные исключения, чтобы повысить читаемость и реализовать обработку специфических исключений. Если вы перехватываете общие исключения `Exception`, то они также включают в себя `RuntimeException`. Некоторые IDE могут самостоятельно генерировать блоки `catch`, которые являются слишком обобщенными. Поэтому вам стоит подумать о том, чтобы их конкретизировать.

Документируйте исключения

Документируйте исключения, включая непроверяемые, на уровне API. Это поможет при устранении неисправностей. На самом деле непроверяемые исключения в отчете указывают на источник проблемы, который потом можно найти. Пример 3-22 демонстрирует документирование исключений с использованием ключевых слов `@throws` синтаксиса Javadoc.

Пример 3-22. Документирование исключений

```
@throws  NoSuchFileException if the file does not exist
@throws  DirectoryNotEmptyException if the file is a directory and could not
otherwise be deleted because the directory is not empty
```

```
@throws  IOException if an I/O error occurs
@throws  SecurityException In the case of the default provider, and a security
manager is installed, the {@link SecurityManager#checkDelete(String)} method is
invoked to check delete access to the file
```

Будьте осторожны с исключениями, связанными с конкретной реализацией

Не создавайте исключения, связанные с конкретной реализацией, потому что это нарушает принцип инкапсуляции вашего API. Объявление метода `read()` в примере 3-23 принуждает любые его будущие реализации генерировать исключение `OracleException`, при том, что метод `read()` работает с источниками данных, абсолютно не связанными с Oracle.

Пример 3-23. Избегайте исключений, зависящих от конкретной реализации

```
public String read(final Source source) throws OracleException { ... }
```

Исключения против управляющего потока

Не используйте исключения для управления потоком. Пример 3-24 демонстрирует ситуацию, когда код полагается на исключение для выхода из цикла чтения.

Пример 3-24. Использование исключений для управления потоком

```
try {
    while (true) {
        System.out.println(source.read());
    }
}
catch(NoDataException e) {
}
```

Такой ситуации следует избегать по нескольким причинам. Во-первых, ухудшается восприятие кода, потому что синтаксис конструкции `try\catch` создает некий беспорядок. Во-вторых, содержимое самого кода становится менее понятным. Исключения подразумевают работу с ошибками и непредвиденными ситуациями. Следовательно, не стоит создавать исключение, если вы не уверены в том, что оно необходимо. И, наконец, при генерации исключений могут возникать дополнительные проблемы с трассировками стека.

Альтернативы исключениям

Вы узнали, как использовать исключения в Java, чтобы сделать свой анализатор банковских операций более устойчивым и понятным для пользователей. Но есть ли какие-то альтернативы исключениям? Сейчас мы кратко расскажем о четырех альтернативах и об их плюсах и минусах.

Использование null

Вместо того чтобы создавать индивидуальное исключение, почему нельзя просто вернуть значение null, как показано в примере 3-25?

Пример 3-25. Возврат null вместо исключения

```
final String[] columns = line.split(",");

if(columns.length < EXPECTED_ATTRIBUTES_LENGTH) {
    return null;
}
```

Такого подхода однозначно нужно избегать. На самом деле значение null не дает абсолютно никакой полезной информации. При этом подобная конструкция подвержена ошибкам, поскольку возникает необходимость тщательно проверять результат работы API на не-null результат. На практике это приводит к большому количеству исключений NullPointerExceptions и длительному процессу отладки приложения.

Шаблон null-объекта

В Java иногда можно встретить такой подход, как *шаблон null-объекта*. Если говорить коротко, то вместо возврата нулевой ссылки, которая указывает на отсутствие объекта, вы возвращаете некий объект, который реализует ожидаемый интерфейс, но метод внутри него пуст. Преимущество такого подхода состоит в том, что вам не приходится сталкиваться с исключениями NullPointerExceptions и с огромным количеством проверок на null. По факту пустой объект довольно предсказуем, потому что не несет никакого функционала. Однако такой шаблон тоже может быть проблемным, потому что вы рискуете скрыть потенциальные проблемы с данными за счет объекта, который просто их игнорирует. Как следствие, значительно усложняется отладка.

Optional<T>

В Java 8 появился встроенный тип данных `java.util.Optional<T>`, который предназначен для информирования о наличии или отсутствии значения. `Optional<T>` поставляется с набором методов для обработки отсутствия значений, что уменьшает количество багов. Также у вас есть возможность «соединять» несколько объектов `Optional` и использовать их в качестве возвращаемого значения из различных API. Пример такого использования — метод `findAny()` в Streams API. Более подробно об использовании `Optional<T>` мы расскажем в главе 7.

Try<T>

Есть и другой тип данных — `Try<T>`. Он представляет собой операцию, которая может быть либо удачной, либо нет. Фактически это аналог `Optional<T>`, только в первом случае вы работаете со значениями, а во втором — с операциями. Другими словами, тип данных `Try<T>` дает похожие преимущества в компоновке кода, а также снижает количество ошибок. Как ни странно, тип `Try<T>` не встроен в JDK, но поддерживается сторонними библиотеками, которые легко найти.

Использование сборщиков

Вы уже познакомились с основными принципами правильного программирования. Но что насчет структурирования, сборки и запуска приложений? Из этого раздела вы узнаете, почему необходимо пользоваться сборщиками и как работать с такими сборщиками, как Maven и Gradle, чтобы собирать и запускать приложения определенным образом. В главе 5 вы более детально познакомитесь с сопряженной темой: как эффективно структурировать приложение с помощью пакетов.

Зачем нужны сборщики

Предлагаем решить проблему запуска приложения. Есть несколько моментов, о которых нужно позаботиться. Во-первых, после написания самого кода его необходимо скомпилировать. Чтобы это сделать, можно воспользоваться Java-компилятором (`javac`). Вы помните все нужные команды для компилирования нескольких файлов? Как насчет пакетов? Что вы скажете о зависимостях при импорте сторонних библиотек? А что, если проект

нужно упаковать в специальный формат вроде WAR или JAR? Согласитесь, все быстро запутывается, и разработчику становится все сложнее и сложнее в этом разобраться.

Чтобы автоматизировать все команды, вам придется написать скрипт. Так вам не понадобится каждый раз вводить их заново. Создание нового скрипта требует того, чтобы все ваши настоящие и будущие коллеги хорошо представляли ваш образ мышления и имели возможность обслуживать и модернизировать скрипт по мере необходимости. Кроме того, нужно учитывать жизненный цикл программного обеспечения. И это касается не только разработки и компиляции, но также тестирования и развертывания.

Решение всех этих проблем — использование сборщиков. Инструмент для сборки — это ваш помощник, который выполняет повторяющиеся действия в течение жизненного цикла программного обеспечения, включая разработку, тестирование и развертывание приложения. У сборщиков есть много плюсов:

- Они позволяют придерживаться общей структуры проектов, так что ваши коллеги могут сразу почувствовать себя как дома.
- Они настраивают вас на типовой и стандартизированный процесс сборки и запуска приложения.
- Вы тратите больше времени на разработку, а не на низкоуровневые настройки.
- Снижается количество ошибок, возникающих из-за неправильной настройки или пропущенных команд при сборке.
- Вы экономите время за счет повторного использования задач по сборке.

Сейчас вы познакомитесь с двумя наиболее популярными в Java-сообществе сборщиками: Maven и Gradle[1].

Работа с Maven

Maven наиболее популярен в Java-сообществе. Он позволяет описать процесс сборки вашего программного обеспечения вместе с зависимостями. Кроме того, есть большой репозиторий сообщества, который Maven может

[1] Раньше у Java был другой популярный сборщик под названием `Ant`, но сейчас он считается «умершим» и больше не может быть использован. — *Прим. авт.*

использовать для автоматической загрузки библиотек и зависимостей. Первоначально Maven был представлен в 2004 году, когда, как вы можете догадаться, был очень популярен XML. Следовательно, объявление процесса сборки в Maven основано на XML.

Структура проекта

Самое прекрасное в Maven — это то, что с самого начала он поставляется со структурой, помогающей в обслуживании. Проект Maven начинается с двух основных папок:

`/src/main/java`

Здесь вы сможете найти все классы, необходимые для вашего проекта.

`src/test/java`

Здесь должны располагаться все ваши тесты.

Есть еще две дополнительные папки, удобные, но не обязательные:

`src/main/resources`

Здесь вы можете располагать дополнительные ресурсы вашего проекта, такие как текстовые файлы.

`src/test/resources`

Здесь вы можете располагать дополнительные ресурсы для тестов.

Применение такой схемы расположения файлов позволяет любому человеку, знакомому с Maven, сразу же найти нужные файлы. Чтобы специализировать процесс сборки, вам нужно создать XML-файл, в котором указываются необходимые объявления, задающие порядок сборки приложения. На рис. 3-2 показана типовая структура проекта Maven.

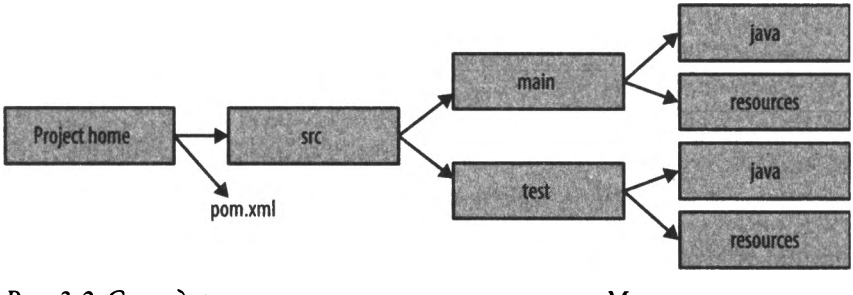

Рис. 3-2. Стандартная схема расположения папок Maven

Пример сборочного файла

Следующий шаг — создание файла *pom.xml*, управляющего процессом сборки. Фрагмент кода, приведенный в примере 3-26, демонстрирует базовый набор, необходимый для сборки проекта анализатора банковских операций. В этом файле вы найдете несколько элементов.

`project`

Это элемент верхнего уровня во всех файлах *pom.xml*.

`groupId`

Этот элемент показывает уникальный идентификатор организации, создавшей проект.

`artifactId`

Этот элемент указывает уникальное базовое имя для артефакта, полученного в процессе сборки.

`packaging`

Этот элемент указывает тип пакета, который используется данным артефактом (такой как JAR, WAR, EAR и т. д.). Если ничего не указано, то по умолчанию применяется тип JAR.

`version`

Версия артефакта, сгенерированного из проекта.

`build`

Этот элемент указывает на различные конфигурации, применяемые в процессе сборки, такие как плагины и ресурсы.

`dependencies`

Этот элемент определяет список зависимостей в проекте.

Пример 3-26. Файл сборки pom.xml в Maven

```
<?xml version="1.0" encoding="UTF-8"?>
<project xmlns="http://maven.apache.org/POM/4.0.0"
         xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
         xsi:schemaLocation="http://maven.apache.org/POM/4.0.0 http://maven.apache.
org/xsd/maven-4.0.0.xsd">
    <modelVersion>4.0.0</modelVersion>
```

```
    <groupId>com.iteratrlearning</groupId>
    <artifactId>bankstatement_analyzer</artifactId>
    <version>1.0-SNAPSHOT</version>

    <build>
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.7.0</version>
                <configuration>
                    <source>9</source>
                    <target>9</target>
                </configuration>
            </plugin>
        </plugins>
    </build>

    <dependencies>
        <dependency>
            <groupId>junit</groupId>
            <artifactId>junit</artifactId>
            <version>4.12</version>
            <scope>test</scope>
        </dependency>
    </dependenciesn>
</project>
```

Команды Maven

Следующий шаг после настройки *pom.xml* — непосредственно использование Maven для сборки проекта. Есть несколько доступных команд. Мы расскажем только об основных:

```
mvn clean
```

Очищает все предыдущие сгенерированные артефакты в предварительной сборке.

```
mvn compile
```

Компилирует исходный код проекта (по умолчанию в созданной папке *target*).

`mvn test`

Тестирует скомпилированный исходный код.

`mvn package`

Запаковывает скомпилированный код в подходящий формат вроде JAR.

Например, выполнение команды `mvn package` из директории, в которой расположен файл *pom.xml*, на выходе дает примерно такой результат:

```
[INFO] Scanning for projects...
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] Building bankstatement_analyzer 1.0-SNAPSHOT
[INFO] ------------------------------------------------------------------------
[INFO]
[INFO] ------------------------------------------------------------------------
[INFO] BUILD SUCCESS
[INFO] ------------------------------------------------------------------------
[INFO] Total time: 1.063 s
[INFO] Finished at: 2018-06-10T12:14:48+01:00
[INFO] Final Memory: 10M/47M
```

Вы увидите сгенерированный JAR-файл *bankstatement_analyzer-1.0-SNAPSHOT.jar* в папке *target*.

Если вы хотите запустить главный класс в сгенерированном артефакте с помощью команды `mvn`, вам придется познакомиться с плагином `exec`.

Использование Gradle

Maven — не единственный инструмент сборки, доступный в Java. Достаточно популярен также сборщик Gradle. Вы можете спросить: зачем нам еще один инструмент для сборки? Разве Maven не самый распространенный? Дело в том, что у Maven есть один серьезный недостаток. Это использование XML-файлов, делающих его не самым удобным в работе. Приведем пример. Во время рабочего процесса часто бывает нужно обеспечить реализацию пользовательских системных команд, таких как копирование

или перемещение файлов. Определение данных команд в синтаксисе XML совершенно неестественно. Кроме того, XML зарекомендовал себя как «многословный» язык, что может сильно ухудшить обслуживаемость. При этом Maven предлагает множество хороших идей вроде стандартизации структуры проекта, которыми вдохновляется и Gradle. Одним из преимуществ Gradle является использование им дружелюбного языка DSL (Domain Specific Language — предметно-ориентированный язык), использующего языки Groovy или Kotlin для задания параметров процесса сборки. В результате параметризация сборки проходит более естественно, она проще настраивается и легче воспринимается. Кроме того, Gradle поддерживает такие полезные особенности как кэш и инкрементная компиляция, которые способствуют сокращению времени сборки[1].

Пример сборочного файла

Структура проекта в Gradle довольно проста и напоминает Maven. Однако вместо файла *pom.xml* вам нужно создать файл *build.grade*. Также есть и файл *settings.gradle*, в котором находятся конфигурационные переменные и настройки для мультипроектной сборки. В примере 3-27 показан фрагмент кода простого сборочного файла Gradle, эквивалентного файлу Maven из примера 3-26. Вы должны признать, что все выглядит гораздо более лаконично!

Пример 3-27. Файл сборки build.gradle в Gradle

```
apply plugin: 'java'
apply plugin: 'application'

group = 'com.iteratrlearning'
version = '1.0-SNAPSHOT'

sourceCompatibility = 9
targetCompatibility = 9

mainClassName = "com.iteratrlearning.MainApplication"

repositories {
    mavenCentral()
}
```

[1] С более подробным сравнением сборщиков Maven и Gradle можно ознакомиться на gradle.org/maven-vs-gradle. — *Прим. авт.*

```
dependencies {
    testImplementation group: 'junit', name: 'junit', version:'4.12'
}
```

Команды Gradle

Наконец, можно запустить процесс сборки путем выполнения команд, похожих на те, что были в Maven. Каждая команда в Maven — это задача. Вы можете определять свои собственные задачи и затем выполнять их, или пользоваться встроенными задачами, такими как `test`, `build` и `clean`:

`gradle clean`

Очищает файлы, созданные во время предыдущей сборки.

`gradle build`

Упаковывает приложение.

`gradle test`

Запускает тесты.

`gradle run`

Запускает главный класс, указанный в поле `mainClassName`, с учетом примененных плагинов.

Ниже представлен пример результата выполнения команды `gradle build`:

```
BUILD SUCCESSFUL in 1s

2 actionable tasks: 2 executed
```

Сгенерированный JAR-файл вы найдете в папке *build*, созданной Gradle в процессе сборки.

Выводы

- Принцип открытости/закрытости основан на идее возможности изменения поведения метода или класса без необходимости изменения его кода.
- За счет неизменяемости существующего кода принцип открытости/закрытости снижает его хрупкость. Принцип продвигает идею

многократного использования существующего кода и «развязывания», что способствует улучшению обслуживаемости.

- Огромные интерфейсы с большим количество специфических методов ведут к повышению сложности и связанности.
- Слишком маленькие интерфейсы с одним методом могут ухудшить показатель связности.
- Вам не стоит волноваться на счет добавления «описательных» имен методов, чтобы повысить читаемость и восприятие вашего API.
- Операции, возвращающие в результате `void`, сложно протестировать.
- Исключения в Java способствуют документированию, безопасности типов и разделению задач.
- Используйте проверяемые исключения более умеренно, так как они могут стать причиной приличной путаницы.
- Слишком специфические (индивидуальные) исключения могут сделать программирование неэффективным.
- Шаблон уведомления позволяет доменному классу накапливать ошибки.
- Не игнорируйте исключения или не перехватывайте общие исключения (generic-исключения), потому что так вы лишите себя возможности быстрого поиска источника проблемы.
- Сборщики автоматизируют повторяющиеся задачи, возникающие в течение жизненного цикла программного продукта, включая сборку, тестирование и развертывание приложения.
- Maven и Gradle — два наиболее популярных в сообществе Java сборщика.

Самостоятельная работа

Если вы хотите расширить или укрепить знания, полученные в этой главе, можете попробовать сделать что-нибудь из следующего:

- Реализовать поддержку экспорта в различные форматы данных, включая JSON и XML.
- Разработать базовый графический интерфейс (GUI) для анализатора банковских операций.

В завершение

Марк Эрбергцук очень доволен вашей последней версией анализатора банковских операций. Через несколько дней после завершения проекта мир накроет новый экономический кризис, и ваше приложение станет очень популярным. Настало время приступить к новому увлекательному проекту, который ждет вас в следующей главе!

Система управления документами

Задача

После успешной реализации анализатора банковских операций для Марка Эрбергцука вы решили заняться своими личными делами. В том числе сходить на прием к стоматологу. Доктор Авадж успешно работает уже много лет. Зубы ее счастливых пациентов остаются здоровыми до самой старости. Единственный минус в такой работе — это то, что с каждым годом количество документации на пациентов только растет. Каждый раз ее помощники тратят все больше времени на поиски записей о предыдущем посещении пациента.

Она понимает, что, похоже, пришла пора заняться автоматизацией процесса управления документами, чтобы содержать их в порядке. К счастью, у нее есть пациент, который может помочь с этим. Вы напишете программу для управления и хранения всех документов, что облегчит поиск нужной информации и сделает работу доктора более продуктивной.

Цель

В этой главе вы откроете для себя много новых принципов программирования. Ключ к разработке системы управления документами — это наследственные связи, подразумевающие наследование классов или реализацию интерфейсов. Чтобы сделать все правильно, вы познакомитесь с принципом подстановки Лисков (Liskov substitution principle, LSP), названным в честь известного ученого в области компьютерных технологий Барбары Лисков.

Затем вы дополните свои знания о том, когда стоит применять наследование, обсуждением принципа «Композиция вместо наследования». И наконец, вы глубже изучите вопрос создания автоматизированных тестов.

Итак, узнав, что вас ожидает в этой главе, вы можете приступить к рассмотрению требований доктора Авадж к системе управления документами.

Если в какой-то момент вам захочется взглянуть на исходный код, вы можете найти его в репозитории по адресу: `/com/iteratrlearning/shu_book/chapter_04`.

Требования к системе управления документами

За чашечкой чая доктор Авадж пояснила, что у нее есть документы, которые она хотела бы систематизировать в виде файлов на компьютере. Система управления документами должна иметь возможность импортировать данные файлы и создавать определенные записи о каждом из них. Доктор хочет, чтобы эти записи можно было индексировать и осуществлять по ним поиск. Ее волнует три типа документов:

Отчеты

Текст с пояснениями по лечению пациента.

Письма

Текстовые документы, которые куда-то отправляются. (Возможно, вы уже сталкивались с чем-то таким, подумайте...)

Изображения

В стоматологической практике часто присутствуют рентгеновские снимки или фотографии зубов и десен. Они имеют определенный размер.

Кроме того, вся документация должна быть «привязана» (прописан путь) к файлам пациентов, к которым она относится. Доктор Авадж хочет иметь возможность поиска документов и запрашивать, содержится ли определенная информация в различных типах документов. Например, найти письма, в которых встречается фраза «Джо Блоггс».

Во время беседы вы подметили, что доктор Авадж планирует в будущем внедрять и другие типы документов.

Воплощение идеи

Для решения поставленной задачи можно воспользоваться большим количеством различных методов. Все они относительно субъективные. Поэтому мы предлагаем вам попробовать решить задачу доктора Авадж дважды — до и после прочтения данной главы. Из раздела «Альтернативные подходы» вы узнаете, почему мы стараемся избегать некоторых методов, а также познакомитесь с общими принципами, лежащими в их основе. Работу над любым приложением лучше всего начинать с разработки через тестирование (TDD — Test-Driven Development), чем мы и руководствовались при написании примеров. Мы не будем рассматривать TDD до главы 5, поэтому давайте просто хорошенько подумаем о том, как должна вести себя ваша программа, и постепенно, шаг за шагом, напишем код, реализующий это поведение.

Система управления документами должна при необходимости импортировать документы и добавлять их в свое внутреннее хранилище. Для того чтобы реализовать данное требование, давайте создадим класс `DocumentManagementSystem` и добавим в него два метода:

```
void importFile(String path)
```

Получает путь к файлу, который пользователь хочет импортировать в систему управления документами. Поскольку мы имеем дело с публичным API, который может принимать пользовательский ввод, в качестве типа данных для пути к файлу мы воспользуемся типом `String`, вместо более безопасных типов вроде `java.nio.Path` или `java.io.File`.

```
List<Document> contents()
```

Возвращает список документов, которые в настоящее время хранит система управления документами.

Вы заметили, что метод `contents()` возвращает список объектов класса `Document`. Пока мы еще не обсуждали назначение этого класса, но будем говорить о нем в течение курса. А пока вы можете считать, что это пустой класс.

Импортеры

Основной характеристикой системы является возможность импорта документов различного типа. В рамках нашей программы для определения порядка импорта документов вы можете полагаться на их формат, поскольку доктор Авадж сохраняет свои файлы в конкретных расширениях. Все ее письма имеют расширение *.letter*, отчеты — *.report*, а для изображений используется только формат *.jpg*.

Проще всего было бы реализовать весь механизм импорта файлов в одном методе (пример 4-1).

Пример 4-1. Пример «переключателя расширений»

```
switch(extension) {
    case "letter":
        // code for importing letters.
        break;

    case "report":
        // code for importing reports.
        break;

    case "jpg":
        // code for importing images.
        break;

    default:
        throw new UnknownFileTypeException("For file: " + path);
}
```

Такой подход однозначно решает поставленную проблему, но его достаточно сложно реализовать. Каждый раз при добавлении нового типа файла вам придется модернизировать конструкцию оператора `switch`. Спустя какое-то время данный метод может стать ужасно длинным и плохо читаемым.

Если вы сохраните свой главный класс простым и отделите от него всевозможные классы-импортеры документов, вам будет значительно проще понимать, какой класс за что отвечает. А чтобы реализовать поддержку различных типов документов, можно объявить интерфейс `Importer`. Каждый `Importer` будет классом, предназначенным для импорта определенного типа файлов.

Теперь, когда мы знаем, что для импорта файлов нам нужен интерфейс, возникает вопрос, в каком виде представлять файлы, подлежащие импорту? У нас есть два варианта: использовать простой тип `String` для представления пути к файлу или воспользоваться классом для работы с файлами вроде `java.io.File`.

Возможно, вы предпочтете вариант, соответствующий принципу жесткой типизации: выберете тип, представляющий файлы и снижающий риск возникновения ошибок. Давайте так и поступим и воспользуемся объектом `java.io.File` в качестве параметра для нашего интерфейса `Importer` (пример 4-2).

Пример 4-2. `Importer`

```
interface Importer {
    Document importFile(File file) throws IOException;
}
```

Вы можете спросить, *почему мы не можем так же использовать `File` и для публичного API `DocumentManagementSystem`?* Потому что в случае с нашим приложением API, возможно, будет «обернут» в какой-нибудь пользовательский интерфейс, и мы не знаем наверняка, в каком виде тот будет принимать файлы. Поэтому мы, чтобы ничего не усложнять, просто используем тип `String`.

Класс `Document`

Теперь давайте объявим класс `Document`. Предполагается, что каждый документ будет иметь несколько атрибутов, по которым должен осуществляться поиск. У разных документов разный набор атрибутов. В данном случае у нас опять же есть несколько вариантов, плюсы и минусы которых мы рассмотрим при создании класса.

Самым простым способом представления документов было бы использование интерфейса `Map<String,String>`, подразумевающего хранение данных в виде пар ключ/значение. Так почему же нам не воспользоваться этим способом и просто не передавать `Map<String, String>` по всему приложению? Что ж, внедрение доменного класса для моделирования документа — это не просто слепое следование принципам ООП, но также и получение ряда практических преимуществ в обслуживаемости и читаемости программы.

Для начала скажем, что невозможно переоценить значение присвоения конкретных имен компонентам программы. Коммуникация — наше все! Хорошие команды разработчиков используют некий *единый язык* для описания своего программного обеспечения. Если словарный запас, которым вы пользуетесь при написании приложения, совпадает со словарным запасом, который вы используете при общении с клиентами вроде доктора Авадж, — вам становится гораздо проще работать с приложением. В процессе разговора с коллегой или с клиентом вам непременно придется прийти к какой-то общей терминологии, при помощи которой вы будете описывать приложение. Если вы примените то же правило к коду, вам будет значительно легче понять, с какой частью кода нужно работать. Это называется *открытостью*.

Понятие «*единого языка*» было предложено Эриком Эвансом и берет свое начало в *предметно-ориентированном программировании*. Оно подразумевает использование общего языка, разработанного для общения между разработчиками и пользователями.

Еще один принцип, который должен побудить вас внедрить дополнительный класс для создания модели документа — это жесткая типизация. Многие относятся к данному понятию как к свойству языка программирования, но здесь мы говорим о более практической стороне использования статической типизации в разработке программного обеспечения. Типы позволяют нам ограничить область использования данных. К примеру, наш класс `Documents` является неизменяемым. После его создания вы не можете в нем что-то *изменить*, или *модернизировать* какой-то из его атрибутов. Реализации нашего интерфейса `Importer` создают документы. Их больше ничего не изменяет. Если вы вдруг обнаруживаете экземпляр `Document` с ошибкой в одном из его атрибутов, то можете легко сузить круг подозреваемых реализаций `Importer`. Еще один плюс неизменяемости: вы можете индексировать или кэшировать любую информацию, связанную с экземплярами `Document` и не сомневаться в том, что эта информация всегда будет корректной, так как документы неизменяемы.

Другой вариант, который могут выбрать разработчики при проектировании `Document`, — это сделать его расширением `HashMap<String, String>`. Поначалу идея кажется прекрасной, поскольку в `HashMap` имеется весь необходимый функционал для моделирования `Document`. Однако есть несколько причин считать эту идею плохой.

Разработка программного обеспечения часто подразумевает ограничение функциональности чего-либо с целью достижения желаемого результата. Мы просто выбросили бы на помойку все вышеперечисленные преимущества неизменяемости, позволив чему угодно в приложении изменять класс `Document`, если бы мы сделали его подклассом `HashMap`. Использование коллекций также дает нам возможность давать методам «значащие» имена вместо того, чтобы, к примеру, искать атрибут через вызов метода `get()` — что, по сути, ни о чем нам не говорит. Чуть позже мы подробно поговорим о конкуренции между наследованием и композицией, потому что сейчас мы имеем дело с прекрасным примером, подходящим для обсуждения данной темы.

Если говорить коротко, доменные классы позволяют именовать элементы и ограничивать рамки поведения и изменения значений этих элементов, что повышает открытость («понятность») кода, снижает вероятность багов. Итак, в конце концов мы принимаем решение моделировать `Document` так, как показано в примере 4-3. Возможно, вы удивлены, что тип класса не `public`, мы обсудим это позже в разделе «Выбор области действия и инкапсуляции».

Пример 4-3. Document

```
public class Document {
    private final Map<String, String> attributes;

    Document(final Map<String, String> attributes) {
        this.attributes = attributes;
    }

    public String getAttribute(final String attributeName) {
        return attributes.get(attributeName);
    }
}
```

Следует обратить внимание на еще один момент касательно `Document`: у него пакетный конструктор. Обычно, классы в Java имеют конструкторы типа `public`, однако в данном случае это может оказаться плохим решением, поскольку позволит коду в любом месте программы создавать объекты такого типа. Только код в системе управления документами должен иметь право создавать объекты `Document`, поэтому мы делаем конструктор пакетным и ограничиваем доступ, предоставляя его только тому пакету, в котором находится наша система управления документами.

Атрибуты и иерархия Documents

В классе Document мы используем атрибуты типа String. Разве это не противоречит принципам строгой типизации? И да и нет. Мы храним атрибуты в виде текста, так что по ним можно делать поиск. Кроме того, мы должны быть уверены, что все атрибуты созданы в правильной изначально заданной форме, не зависящей от создавшего их импортера. Применение String — не такое уж и плохое решение в данном случае. Стоит помнить, что передавать String в приложении с целью представления информации — определенно плохая идея. Это уже не строгая, а очень жесткая типизация.

В частности, если имеет место более сложное использование атрибутов, в таком случае можно подумать об использовании атрибутов различного типа. К примеру, если бы мы захотели найти адреса, расположенные на определенном расстоянии, или изображения, ширина и высота которых меньше заданной, тогда наличие строго типизированных атрибутов было бы кстати. Гораздо проще сравнивать значение ширины, если оно представлено как целое число. В случае с нашей системой управления документами нам просто не нужен такой функционал.

Можно спроектировать систему управления документами с классовой иерархией Documents, которая моделировала бы иерархию Importer. Например, ReportImporter импортирует объекты класса Report, которые являются расширением класса Document. Такой подход проходит нашу стандартную проверку на разумность введения подклассов. Другими словами, данная схема позволяет утверждать, что Report — это Document, и это утверждение имеет смысл. Однако мы решили не идти этим путем, поскольку правильным подходом при конструировании классов в ООП будет учитывать и его поведение, и данные.

Все документы моделируются весьма обобщенно в рамках названных атрибутов, в отличие от конкретных полей, которые существуют в рамках конкретных подклассов. Кроме того, что касается этой системы, документы особо не обладают каким-либо поведением, предусмотренным для них. Нет никакого смысла добавлять иерархию классов, если она не приносит пользы. Вы можете подумать, что это утверждение само по себе является неким критерием, но на самом деле оно говорит нам о другом принципе: KISS.

Вы изучали принцип KISS в главе 2. KISS говорит нам о том, что разработка тем лучше, чем она проще. Часто бывает очень трудно избежать излишней сложности, но работа в данном направлении стоит того. Когда кто-нибудь

говорит: «Нам может понадобиться X» или «было бы хорошо сделать Y» — просто скажите: «Нет». Раздутые и сложные конструкции вымощены благими намерениями, направленными на расширяемость и создание кода, который «приятно иметь», а не «обязательно иметь».

Реализация и регистрация импортеров

Вы можете реализовать интерфейс `Importer` с целью поддержки различных типов файлов. В примере 4-4 показан способ импорта изображений. Одним из больших преимуществ стандартной библиотеки Java является то, что она предоставляет очень много функциональных возможностей прямо «из коробки». Здесь мы считываем файл изображения при помощи метода `ImageIO.read`, а затем извлекаем ширину и высоту этого изображения из результирующего объекта `BufferedImage`.

Пример 4-4. ImageImporter

```
import static com.iteratrlearning.shu_book.chapter_04.Attributes.*;

class ImageImporter implements Importer {
    @Override
    public Document importFile(final File file) throws IOException {
        final Map<String, String> attributes = new HashMap<>();
        attributes.put(PATH, file.getPath());

        final BufferedImage image = ImageIO.read(file);
        attributes.put(WIDTH, String.valueOf(image.getWidth()));
        attributes.put(HEIGHT, String.valueOf(image.getHeight()));
        attributes.put(TYPE, "IMAGE");

        return new Document(attributes);
    }
}
```

Имена атрибутов заданы в виде констант в классе `Attributes`. Это исключает баги в случае, если разные импортеры будут использовать различные строки для одного и того же имени атрибута. Например, «Path» вместо «path». В самом языке Java нет непосредственно понятия константы как таковой, пример 4-5 показывает наиболее часто используемый прием. В данном случае константа имеет тип `public`, потому что мы хотим использовать ее из

разных импортеров, хотя вы могли бы использовать `private` или даже пакетную константу. Ключевое слово `final` дает уверенность в том, что значение константы не будет переопределено, а `static` — в том, что у нее может быть только один экземпляр на класс.

Пример 4-5. Как объявить константу в Java

```
public static final String PATH = "path";
```

Существуют импортеры для всех трех типов файлов, но оставшиеся два вы увидите в разделе «Расширение и повторное использование кода». Не волнуйтесь, мы ничего не прячем в рукавах. Вообще, для использования классов `Importer` в процессе импорта файлов нам нужно зарегистрировать импортеры, чтобы увидеть их. Мы используем расширение файла, который хотим импортировать, в качестве ключа для объекта `Map` (пример 4-6).

Пример 4-6. Регистрация импортеров

```
private final Map<String, Importer> extensionToImporter = new HashMap<>();

public DocumentManagementSystem() {
    extensionToImporter.put("letter", new LetterImporter());
    extensionToImporter.put("report", new ReportImporter());
    extensionToImporter.put("jpg", new ImageImporter());
}
```

Теперь, когда вы знаете, как импортировать документы, можно заняться поиском. Мы не задаемся целью создать самый эффективный поиск хотя бы потому, что не создаем Google. Нам нужно просто получать информацию, которую запрашивает доктор Авадж. Из разговора с ней вы поняли, что она хочет иметь возможность просматривать информацию о разных атрибутах `Document`.

Требования доктора Авадж можно выполнить, просто учитывая последовательности атрибутов. Допустим, ей понадобится найти документы, относящиеся к пациенту по имени Джо и содержащие упоминание *диетической колы* в тексте. Для этого мы разработали очень простой язык запросов. Запросы состоят из последовательности имен атрибутов и значений, разделенных запятыми. Такой запрос выглядит примерно так: `"patient:Joe,body:Diet Coke"`.

Поскольку наш алгоритм поиска должен оставаться простым, а не максимально эффективным, то он просто последовательно сканирует все записи в системе и проверяет их на соответствие запросу.

Строка запроса передается методу `search`, преобразовывается в объект `Query`, который потом можно тестировать на соответствие каждому экземпляру `Document`.

Принцип подстановки Лисков

Мы обсудили некоторые определенные решения в разработке, касающиеся классов. Например, мы рассмотрели вопрос моделирования импортеров с помощью классов, поговорили о том, почему не стоит вводить иерархию классов для класса `Document` или почему лучше не делать `Document` просто расширением `HashMap`. На самом деле все это подводит нас к более важному принципу. К принципу, позволяющему обобщить указанные выше примеры и объединить их в один подход, который вы можете применять в любом программном продукте. Он называется *принципом подстановки Лисков* (Liskov Substitution Principle — LSP) и помогает нам понять, как правильно организовывать связь и реализовывать интерфейсы. LSP — третий из принципов SOLID, к которым мы будем обращаться на протяжении всей книги.

Принцип подстановки Лисков часто формулируется в довольно сложных формальных терминах, но на самом деле он очень прост. Давайте немного проясним терминологию. В данном контексте встречаясь со словом *тип*, думайте о классе или об интерфейсе. Термин *подтип* подразумевает установленные наследственные отношения (родитель — ребенок) между типами. Другими словами, является расширением класса или реализацией интерфейса. Проще говоря, вы можете считать, что дочерние классы должны реализовывать поведение, которое они наследуют от своих родителей. Знаем, знаем — звучит весьма очевидно, но сейчас мы конкретизируем кое-что и разобьем принцип LSP на четыре части:

> **LSP**
>
> Пусть $q(x)$ — свойство, доказуемое для объектов x типа T. Тогда $q(y)$ должно быть истинно для объектов y типа S, где S — подтип T.

Предварительные условия не могут быть усилены в подтипе

Предусловие устанавливает условия, при которых работает определенный фрагмент кода. Нельзя считать, что ваш код будет работать всегда, везде и при любых условиях. Например, все реализации `Importer` имеют предусловие, проверяющее, что импортируемый файл существует и его можно считать. В результате у метода `importFile` есть проверочный код, который выполняется перед запуском (как показано в примере 4-7).

Пример 4-7. Определение `importFile`

```
    public void importFile(final String path) throws IOException {
        final File file = new File(path);
        if (!file.exists()) {
            throw new FileNotFoundException(path);
        }

        final int separatorIndex = path.lastIndexOf('.');
        if (separatorIndex != -1) {
            if (separatorIndex == path.length()) {
                throw new UnknownFileTypeException("No extension found For file: "
+ path);
            }
            final String extension = path.substring(separatorIndex + 1);
            final Importer importer = extensionToImporter.get(extension);
            if (importer == null) {
                throw new UnknownFileTypeException("For file: " + path);
            }
            final Document document = importer.importFile(file);
            documents.add(document);
        } else {
            throw new UnknownFileTypeException("No extension found For file: " +
path);
        }
    }
```

Принцип LSP подразумевает, что вы не можете требовать больше ограничивающих предусловий, чем в родительском элементе. Так, например, вы не можете требовать в дочернем объекте, чтобы размер файла был меньше 100 КБ, если родительский класс может импортировать документы любого размера.

Постусловия не могут быть ослаблены в подтипе

Это утверждение может ввести вас в замешательство, потому что звучит почти так же, как и первое. Постусловия — это условия, которые должны быть истинны после выполнения определенного кода. Например, после выполнения `importFile()`, если запрашиваемый файл не был поврежден, он должен находиться в списке документов, возвращаемом `contents()`. Таким образом, если родительский элемент производит какие-то действия или возвращает какие-то значения, дочерний элемент должен делать так же.

Инварианты сверхтипа должны быть сохранены в подтипе

Инвариант — это то, что никогда не меняется, как направление движения воды при приливе или отливе. В контексте наследования это означает, что мы должны быть уверены, что любые инварианты, обеспечивающиеся родительским классом, должны обеспечиваться и дочерним классом.

Правило истории

Данный аспект LSP является самым тяжелым для понимания. По сути, дочерний класс не должен допускать изменений состояния, которые были запрещены в родительском классе. Так, в нашем примере есть неизменяемый класс `Document`. После того как он начал существовать, вы не можете удалить, добавить или изменить какой-либо из его атрибутов. Вы не можете также создать подкласс класса `Document` и сделать его изменяемым. Все потому, что любой пользователь родительского класса должен ожидать определенного поведения при вызове методов класса `Document`. Если дочерний класс окажется изменяемым, это может нарушить ожидания пользователя относительно результата вызова упомянутых методов.

Альтернативные подходы

При разработке системы управления документами можно было бы применить множество различных подходов. Сейчас мы рассмотрим некоторые из них, так как они, скорее всего, будут вам полезны. Здесь нет абсолютно неправильных подходов, просто мы считаем, что выбранный нами — самый оптимальный.

Поместить импортер в класс

Вы могли бы создать иерархию классов для импортеров. Тогда вместо интерфейса иерархию воглавлял бы класс. Интерфейсы и классы имеют различный набор возможностей. Например, вы можете реализовать множество интерфейсов, в то время как классы могут содержать поля экземпляров, кроме того, как правило, тела методов также находятся в классах.

В данном случае причиной для организации иерархии послужила бы возможность использования различных импортеров. Вы уже знакомы с нашим мнением о том, почему стоит избегать хрупких наследственных отношений, основанных на классах. Именно поэтому мы считаем, что в этом проекте лучше использовать интерфейсы.

Однако это не значит, что классы вообще не стоит применять. Если вам нужно создать сильные наследственные отношения в домене, который имеет много состояний или много сценариев поведения, то более подходящим будет как раз классовое наследование.

Область действия и инкапсуляция

Если вы уделили достаточно времени изучению кода, то наверняка обратили внимание, что интерфейс `Importer`, его реализации и класс `Query` находятся в области действия пакета. Область действия пакета — это область действия по умолчанию. Поэтому, если вы видите файл класса с `class Query` в начале, вы знаете, что он в области действия пакета, если же вы видите `public class Query`, вы понимаете, что класс имеет публичный доступ. Пакетная область действия подразумевает, что только другие классы в том же пакете могут «видеть» или иметь доступ к данному классу, больше никто не может. Похоже на маскировку.

Странная особенность экосистемы Java: несмотря на то, что областью действия по умолчанию установлена пакетная область действия, каким бы проектом мы ни занимались, мы всегда встречаем больше `public`-классов, чем классов с пакетной видимостью. По идее, по умолчанию должна быть видимость `public`. Однако на самом деле пакетная видимость — довольно удобный инструмент. Он позволяет инкапсулировать некоторые решения в проектировании. За счет правильного применения пакетной видимости вы можете пресечь попытки классов вне пакета получить информацию о внутренней реализации.

В данном разделе мы много обсуждали различные методы и подходы, которые можно было бы применить при разработке системы документации, и, вполне вероятно, вы еще захотите перейти на один из них в процессе обслуживания системы. Напоминаем, что мы просто рассмотрели несколько подходов. Нет ничего изначально неправильного в том, чтобы выбрать какой-то другой из перечисленных в данном разделе вариантов. Они вполне могут оказаться более подходящими, ведь все зависит от того, как приложение будет развиваться с течением времени.

Расширение и повторное использование кода

Что касается программного обеспечения, постоянны только изменения. Через какое-то время вы можете захотеть добавить новые возможности в свой продукт, следуя своему желанию, требованиям заказчика или даже изменениям в правилах и регламентах. Как уже говорилось ранее, доктор Авадж со временем планирует расширить перечень документов, загружаемых в систему. На самом деле когда мы впервые продемонстрировали ей разработанное программное обеспечение, она сразу поняла, что хотела бы иметь возможность выставлять при помощи нашей системы счета клиентам. Счет — это документ с некоторым содержимым (телом) и суммой, имеющий расширение *.invoice*. В примере 4-8 показан пример счета.

Пример 4-8. Пример счета

```
Уважаемый Джо Блоггс!

Это счет за предоставленные вам услуги стоматолога.

Сумма: 100$

С наилучшими пожеланиями,

  Замечательный стоматолог,
  Доктор Авадж.
```

К счастью для нас, все счета доктора Авадж имеют один формат. Как вы понимаете, нам нужно извлечь сумму из текста. Строка с суммой начинается с префикса `Сумма:`. Имя клиента находится в начале письма и начинается со слова `Уважаемый`. По сути, наша система должна реализовать общий метод

поиска суффикса в строке, начинающейся с заданного префикса, как показано в примере 4-9. В данном примере поле `lines` уже было инициализировано и хранит строки из файла, который мы импортировали. Мы передаем методу префикс. Например, `Сумма:`. Он ассоциируется с окончанием строки — суффиксом, с указанным именем атрибута.

Пример 4-9. Определение addLineSuffix

```
void addLineSuffix(final String prefix, final String attributeName) {
    for(final String line: lines) {
        if (line.startsWith(prefix)) {
            attributes.put(attributeName, line.substring(prefix.length()));
            break;
        }
    }
}
```

На самом деле похожую концепцию мы использовали, когда импортировали письмо. Рассмотрим образец письма в примере 4-10. Здесь нам нужно извлечь имя пациента из текста путем поиска строки, начинающейся со слова `Уважаемый`. В письмах также есть адреса и блоки текста, которые нужно извлечь из содержимого текстового файла.

Пример 4-10. Пример письма

```
Уважаемый Джо Блоггс!

Фейк стрит, 123
Вестминстер
Лондон
Великобритания

Мы написали вам это письмо, чтобы подтвердить перенос посещения доктора Авадж с 29
декабря 2016 на 5 января 2017.

С наилучшими пожеланиями,

  Замечательный стоматолог,
  доктор Авадж.
```

Похожая проблема возникает с импортом отчетов о пациентах. Отчеты доктора Авадж перед именем пациента содержат префикс `Пациент:`. Кроме того,

в них есть блоки текста, точно как и в письмах. Образец отчета вы можете увидеть в примере 4-11.

Пример 4-11. Пример отчета

```
Пациент: Джо Блоггс

5 января 2017 я осматривала зубы Джо.
Мы обсуждали его переход с обычной колы на диетическую.
Новых проблем с зубами не обнаружено.
```

Таким образом, все три текстовых импортера могли бы реализовывать один и тот же метод для поиска суффиксов в текстовых строках с заданным префиксом (смотрите пример 4-9). Если бы доктор Авадж платила нам за количество строк написанного кода, мы могли бы утроить сумму, проделав одну и ту же работу три раза! Неплохая стратегия!

К сожалению (а может, и к счастью), заказчики редко платят за количество строк написанного кода. А вот что действительно имеет значение — так это требования заказчика к продукту. Конечно, нам хотелось бы иметь возможность повторно использовать один и тот же код для всех трех импортеров. И это можно сделать, поместив наш код в класс. И тут мы сталкиваемся с тремя возможными вариантами, каждый из которых имеет свои плюсы и минусы. Давайте рассмотим их и подумаем, какой выбрать. Итак, вот наши варианты:

- Использовать *служебный* класс.
- Применить *наследование.*
- Использовать доменный класс.

Самый простой вариант — создать служебный класс. Его можно назвать `ImportUtil`. В таком случае каждый раз, когда вам понадобится метод для использования в разных импортерах, вы сможете обращаться к данному служебному классу. В конечном итоге ваш служебный класс превратится в мешок со статическими методами.

Хотя использование служебного класса — вариант замечательный и простой, его совершенно точно нельзя назвать вершиной объектно-ориентированного программирования. ООП включает в себя концепции моделирования при помощи классов. Если вы хотите создать что-то (thing), то просто вызываете `new Thing()` для того, что вам нужно. Атрибуты и поведение, ассоциированные с этим «чем-то», — это методы класса `Thing`.

Если вы придерживаетесь принципа моделирования реальных объектов в виде классов, это в значительной степени облегчает понимание вашего кода, поскольку предоставляет вам определенную структуру и переносит в код психологическую модель вашего доменного объекта. Хотите изменить алгоритм импорта писем? Хорошо. Просто измените класс `LetterImporter`.

Служебные классы не отвечают подобным ожиданиям и часто приводят к большому количеству процедурного кода, не соответствующего принципу единой ответственности. Со временем это часто «вытекает» в образование классов-богов. Другими словами, вы можете получить один большой класс, который в конечном итоге возьмет на себя очень много функций.

Итак, что же вам делать, если вы хотите, чтобы ваш код соответствовал упомянутым концепциям? Что ж, следующим наиболее очевидным подходом может быть использование наследования. В таком случае у вас будет несколько импортеров, расширяющих класс `TextImporter`. Весь основной функционал вы сможете реализовать в этом классе и повторно использовать его в подклассах.

Наследование — это удивительно устойчивый подход в различных условиях разработки. Вы уже познакомились с принципом подстановки Лисков и ограничениями, которые он накладывает на наследственные отношения. На практике наследование не всегда применимо, потому что зачастую не способно моделировать реальные отношения.

В данном случае `TextImporter` — это `Importer`, и мы можем быть уверены, что наши классы соответствуют принципу LSP, и все же такой подход нельзя назвать лучшим. Проблема наследственных отношений, которые не в полной мере соответствуют отношениям между объектами в реальной жизни, заключается в том, что они имеют тенденцию к хрупкости. Поскольку ваше приложение с течением времени развивается, вам нужны абстракции, способные развиваться вместе с приложением, а не против него. Как правило, не стоит вводить наследование только для того, чтобы обеспечить повторное использование кода.

Последний вариант — смоделировать текстовый файл при помощи доменного класса. Чтобы воспользоваться данным подходом, мы могли бы смоделировать некоторую базовую концепцию и выстроить различные импортеры, вызывая методы поверх базовой концепции. Итак, поскольку мы пытаемся манипулировать содержимым текстовых файлов, давайте назовем класс `TextFile`. Не очень оригинально или креативно, но зато именно

то, что нам нужно. Теперь вы знаете, где находится функционал для работы с текстовыми файлами.

В примере 4-12 показано объявление данного класса и его полей. Обратите внимание, что это не подкласс `Document`, потому что документ не должен быть связан только с текстовыми файлами — ведь мы можем импортировать также бинарные файлы в виде картинок. Мы имеем дело просто с классом, который моделирует базовую концепцию текстового файла и имеет соответствующие методы извлечения данных из текстовых файлов.

Пример 4-12. Объявление класса `TextFile`

```
class TextFile {
    private final Map<String, String> attributes;
    private final List<String> lines;

    // Продолжение кода...
```

Именно такой подход мы применяем в случае с импортерами. Мы считаем, что он позволяет нам достаточно гибко моделировать основную проблему. Он не привязывает нас к хрупкой иерархии наследования, но по-прежнему позволяет повторно использовать код. В примере 4-13 показано, как импортировать счета. Добавлены суффиксы для имени и суммы, а также добавлен тип счета.

Пример 4-13. Импортирование счетов

```
    @Override
    public Document importFile(final File file) throws IOException {
        final TextFile textFile = new TextFile(file);

        textFile.addLineSuffix(NAME_PREFIX, PATIENT);
        textFile.addLineSuffix(AMOUNT_PREFIX, AMOUNT);

        final Map<String, String> attributes = textFile.getAttributes();
        attributes.put(TYPE, "INVOICE");
        return new Document(attributes);
    }
```

Ниже представлен другой пример импортера, который использует класс `TextFile` (пример 4-14). Не стоит волноваться о том, как реализован метод `TextFile.addLines`. Это объясняется в примере 4-15.

Пример 4-14. Импортирование писем

```
    @Override
    public Document importFile(final File file) throws IOException {
        final TextFile textFile = new TextFile(file);

        textFile.addLineSuffix(NAME_PREFIX, PATIENT);

        final int lineNumber = textFile.addLines(2, String::isEmpty, ADDRESS);
        textFile.addLines(lineNumber + 1, (line) -> line.startsWith("regards,"),
BODY);

        final Map<String, String> attributes = textFile.getAttributes();
        attributes.put(TYPE, "LETTER");
        return new Document(attributes);
    }
```

Первоначально данные классы не были написаны именно таким образом. Они эволюционировали и приняли такой вид. Когда мы только приступили к созданию системы управления документами, первый текстовый импортер (`LetterImporter`) имел алгоритм извлечения всего текстового содержимого, написанный прямо в классе. Прекрасный способ начать. Попытки найти код для повторного использования часто приводят к неадекватным абстракциям. Как говорится, прежде чем бегать, нужно научиться ходить.

Когда мы приступили к написанию `ReportImporter`, стало очевидно, что большая часть алгоритма у двух импортеров почти одинакова, и что они действительно должны быть реализованы в рамках вызова методов на основе некоторой базовой концепции, которую мы вам здесь и представили — `TextFile`. По сути, мы скопировали и вставили код, который изначально должен был использоваться двумя классами.

Мы не говорим, что копирование кода — это хорошо. Ни в коем случае. Но иногда при создании новых классов лучше продублировать небольшой фрагмент кода. Когда вы реализуете большую часть приложения, правильная абстракция (такая как `TextFile`) станет очевидной. Идти по пути устранения дублирования стоит только тогда, когда вы знаете, как это правильно сделать.

В примере 4-15 показана реализация метода `TextFile.addLines`. Это общий код, используемый различными реализациями импортеров. Его первый аргумент — индекс `start` — говорит, с какого номера строки начать. Затем

идет элемент isEnd, который применяется к строке и возвращает true, если мы достигли конца строки. В конце идет имя атрибута, которое мы собираемся ассоциировать с этим значением.

Пример 4-15. Объявление addLines

```
int addLines(
    final int start,
    final Predicate<String> isEnd,
    final String attributeName) {

    final StringBuilder accumulator = new StringBuilder();
    int lineNumber;
    for (lineNumber = start; lineNumber < lines.size(); lineNumber++) {
        final String line = lines.get(lineNumber);
        if (isEnd.test(line)) {
            break;
        }

        accumulator.append(line);
        accumulator.append("\n");
    }
    attributes.put(attributeName, accumulator.toString().trim());
    return lineNumber;
}
```

Гигиена тестов

Как вы узнали из главы 2, написание автоматизированных тестов дает много преимуществ с точки зрения обслуживаемости программного обеспечения. Тесты позволяют нам уменьшить количество отказов и понять, что явилось их причиной. Кроме того, они позволяют безопасно модернизировать код. При этом тесты — не панацея, конечно. Чтобы пользоваться всеми «благами» тестов, первоначально нужно их создать, а затем и обслуживать. Как вам известно, написание и обслуживание кода — дело непростое, и многие разработчики отмечают, что создание автоматизированных тестов может занимать немало времени.

Строго говоря, для того чтобы решить проблему поддержки тестов, вам нужно придерживаться определенных правил тестовой гигиены. Тестовая гигиена подразумевает необходимость сохранения кода тестов «чистым»,

а также поддержку и модернизацию тестов вместе с кодом, который они тестируют. Если вы не будете обслуживать и поддерживать свои тесты, через какое-то время они начнут приносить вам проблемы и способствовать снижению продуктивности. В этом разделе вы познакомитесь с некоторыми ключевыми правилами, которые позволят вам поддерживать гигиену ваших тестов.

Именование тестов

Первое, о чем стоит задуматься, когда вы приступаете к работе над тестами — их имена. Порой разработчики становятся слишком самоуверенными в вопросах имен. Это тема, о которой можно говорить вечно, поскольку почти каждый из нас сталкивается с данной проблемой и, конечно, имеет собственное мнение на этот счет. На наш взгляд, важно всегда помнить о том, что где-то есть действительно подходящее имя, и ни в коем случае не забывать о существовании большого количества неудачных вариантов.

Первым тестом, который мы написали для системы управления документами, стал тест, проверяющий, что мы корректно импортируем файл и создаем экземпляр `Document`. Он был написан до того, как мы внедрили концепцию `Importer`, поэтому он не тестирует атрибуты, ориентированные на `Document`. Его код приведен в примере 4-16.

Пример 4-16. Тест импорта файлов

```
@Test
public void shouldImportFile() throws Exception
{
    system.importFile(LETTER);

    final Document document = onlyDocument();

    assertAttributeEquals(document, Attributes.PATH, LETTER);
}
```

Данный тест называется `shouldImportFile`. Ключевыми принципами присвоения имен тестам являются: читаемость, обслуживаемость и выполнение роли *эксплуатационной документации*. Когда вы видите результат работы теста — отчет, имена в нем должны быть похожи на инструкции, документирующие, какой функционал работает, а какой нет. Так разработчик может

легко отслеживать связь между поведением приложения и тестом, который утверждает, что это поведение реализовано. За счет уменьшения несоответствия между поведением и кодом мы облегчаем понимание происходящего в программе и другим разработчикам, которых могут задействовать в проекте в будущем. `shouldImportFile` — это тест, который проверяет алгоритм импорта файлов в системе управления документами.

Стоит отметить, что в именовании существует много антишаблонов. Наихудший из них — назвать тест каким-нибудь совершенно бессмысленным словом, например `test1`. Что тестирует этот `test1`? Терпение читателей? Относитесь к людям, читающим ваш код, так, как вы хотели бы, чтобы они относились к вам.

Другой распространенный антишаблон — называть тесты именами каких-нибудь ключевых элементов вроде `file` или `document`. Имя теста должно описывать тестируемое поведение, а не объект. Еще один антишаблон — назвать тест именем метода, который вызывается в процессе тестирования. Например, `importFile`.

Вы можете спросить: разве, называя тест `shouldImportFile`, мы не грешим тем же самым? В этом обвинении есть доля истины, однако в данном случае мы все-таки описываем поведение, которое тестируется. По сути, метод `importFile` тестируется несколькими тестами. Например, `shouldImportLetterAttributes`, `shouldImportReportAttributes` и `shouldImportImageAttributes`. Ни один из них не называется `importFile` — все они описывают какое-то более конкретное поведение.

Хорошо. Теперь вы знаете, как выглядят плохие имена. А что насчет хороших? При присваивании имен тестам необходимо следовать трем базовым правилам:

Используйте терминологию домена

Старайтесь, чтобы словарный запас, который вы используете при присвоении имен тестам, соответствовал терминологии решаемой задачи или перекликался с самим приложением.

Используйте естественный язык

Любое имя теста должно легко читаться, так же, как обычное предложение. Оно всегда должно описывать какое-то поведение, внятно и понятно.

Старайтесь описывать

Как правило, на чтение кода уходит больше времени, чем на его написание. Не жалейте потратить время на обдумывание правильных названий, которые будут четко описывать содержимое и улучшат читаемость кода в будущем. Если не можете придумать хорошее имя, почему бы не спросить у коллег? В гольфе вы выигрываете, сделав наименьшее количество бросков. В программирование не так. Самый короткий путь — не обязательно самый лучший.

Придумывая имена тестам, вы можете пользоваться принципом, примененным в `DocumentManagementSystemTest` — использовать слово `should` в качестве префикса. А можете и не пользоваться. Это вопрос ваших личных предпочтений.

Поведение, а не реализация

При написании теста для класса, компонента или даже целой системы, необходимо затрагивать только *общедоступное* (*публичное*) *поведение* тестируемого элемента. В случае с системой управления документами мы тестируем только публичный API при помощи теста `DocumentManagementSystemTest`. В данном случае мы тестируем публичный API класса `DocumentManagementSystem` и, таким образом, систему в целом. API представлен в примере 4-17.

Пример 4-17. Публичный API класса `DocumentManagementSystem`

```
public class DocumentManagementSystem
{
    public void importFile(final String path) {
        ...
    }

    public List<Document> contents() {
        ...
    }

    public List<Document> search(final String query) {
        ...
    }
}
```

Наши тесты должны вызывать только методы публичного API и не пытаться при этом проверять внутреннее состояние объектов. Это одна из ключевых

ошибок разработчиков, которая приводит к возникновению сложных в обслуживании тестов. Завязывание теста на мелкие детали делает его хрупким, ведь если в будущем вы измените реализацию этих самых деталей, тест может начать выдавать отрицательные результаты даже при корректном поведении объекта. Давайте рассмотрим тест, приведенный в примере 4-18.

Пример 4-18. Тест импортера писем

```
@Test
public void shouldImportLetterAttributes() throws Exception
{
    system.importFile(LETTER);

    final Document document = onlyDocument();

    assertAttributeEquals(document, PATIENT, JOE_BLOGGS);
    assertAttributeEquals(document, ADDRESS,
        "123 Fake Street\n" +
            "Westminster\n" +
            "London\n" +
            "United Kingdom");
    assertAttributeEquals(document, BODY,
        "We are writing to you to confirm the re-scheduling of your appointment
\n" +
        "with Dr. Avaj from 29th December 2016 to 5th January 2017.");
    assertTypeIs("LETTER", document);
}
```

Одним из способов тестирования алгоритма импортирования писем мог бы стать модульный тест для класса `LetterImporter`. Суть подхода такова: импортировать примерный файл, а затем сделать утверждение касательно результата, возвращаемого импортером. Однако в наших тестах само существование `LetterImporter` является деталью реализации. В разделе «Расширение и повторное использование кода» вы видели ряд вариантов изложения кода импортера. Излагая наши тесты таким же образом, мы оставляем за собой возможность изменить их «наполнение», не нарушая при этом сами тесты.

Итак, мы сказали, что, полагаясь на поведение класса, мы полагаемся на использование публичного API, но обычно у класса есть такие элементы, поведение которых не ограничивается только применением публичных или приватных методов. Например, мы можем не захотеть зависеть от порядка

документов, полученных из метода `contents()`. Это не свойство, которое ограничивается публичным API класса `DocumentManagementSystem`, но то, с чем нужно быть осторожным.

Общим антишаблоном в этом отношении является раскрытие приватного состояния какого-либо объекта за счет геттеров или сеттеров просто для того, чтобы упростить тестирование. Вам стоит пытаться избегать подобного подхода насколько возможно, поскольку он делает ваши тесты хрупкими. Если вы прибегнете к нему, чтобы совсем чуть-чуть упростить тест, в будущем это может привести к большим сложностям при обслуживании. Так происходит потому, что любое изменение кода, предусматривающее изменение отображения внутреннего состояния объекта, требует от вас и изменения теста.

Иногда подобные ситуации служат хорошим индикатором того, что, возможно, вам стоит переработать класс так, чтобы его можно было тестировать легче и эффективнее.

Не повторяйтесь

В разделе «Расширение и повторное использование кода» обсуждается вопрос, как избавиться от дублирования кода в вашем приложении и куда при этом поместить полученный код. Та же самая логика применима и к тестам. К сожалению, разработчики часто не уделяют должного внимания вопросам дублирования в тестах так, как они делают это в основном коде программы. Если вы посмотрите на пример 4-19, то увидите тест, который повторно делает утверждения для различных атрибутов, возвращаемых `Document`.

Пример 4-19. Тест для импорта изображений

```
@Test
public void shouldImportImageAttributes() throws Exception
{
    system.importFile(XRAY);

    final Document document = onlyDocument();

    assertAttributeEquals(document, WIDTH, "320");
    assertAttributeEquals(document, HEIGHT, "179");
    assertTypeIs("IMAGE", document);
}
```

В обычной ситуации вам нужно было бы искать имя каждого атрибута и делать утверждение о соответствии его ожидаемому значению. В случае с приведенными тестами мы имеем дело с достаточно распространенной операцией, когда данную логику реализует общий метод `assertAttributeEquals`. Реализация метода показана в примере 4-20.

Пример 4-20. Реализация нового утверждения

```
private void assertAttributeEquals(
    final Document document,
    final String attributeName,
    final String expectedValue)
{
    assertEquals(
        "Document has the wrong value for " + attributeName,
        expectedValue,
        document.getAttribute(attributeName));
}
```

Хорошая диагностика

Плох тот тест, который не дает отрицательный результат. На самом деле, если вы никогда не видели отрицательного результата, как вы вообще можете знать, что ваш тест работает? При написании тестов лучше всего оптимизировать случаи отрицательного результата. Говоря «оптимизировать», мы не имеем в виду ускорить работу теста при отказе. Мы имеем в виду — убедиться, что тест написан таким образом, что вы четко понимаете, как и в каких случаях он дает отрицательный результат. Секрет этого фокуса — в хорошей *диагностике*.

Под диагностикой мы понимаем сообщение или информацию, которая выводится на экран при провале теста. Чем конкретнее сообщение о том, что пошло не так, тем проще устранить причину отказа. Вы можете спросить, зачем беспокоиться об этом, ведь большинство современных тестов в Java выполняются в современных IDE (средах разработки), в которые уже встроены отладчики? Что ж, иногда тесты могут выполняться в средах непрерывной интеграции, и иногда это может происходить при помощи командной строки. Даже если вы выполняете тестирование из-под IDE — все равно очень полезно иметь полную диагностическую информацию. Надеемся, что мы убедили вас в необходимости качественной диагностики. Но как это выглядит на деле?

В примере 4-21 показан метод, который делает утверждение, что система содержит только один документ. Немного позже мы поясним метод `hasSize()`.

Пример 4-21. Тест, проверяющий, что система хранит только один документ

```
private Document onlyDocument()
{
    final List<Document> documents = system.contents();
    assertThat(documents, hasSize(1));
    return documents.get(0);
}
```

Простейшим видом утверждений, предлагаемым JUnit, является `assertTrue()`, который принимает булево значение и ожидает, что оно истинно. В примере 4-22 показывается, как нужно использовать `assertTrue`, чтобы реализовать тест. В данном случае значение проверяется на эквивалентность нулю. Это приводит к провалу теста `shouldImportFile` и демонстрирует диагностику с отрицательным результатом. Проблема в том, что здесь мы не получаем действительно хорошую диагностическую информацию. Просто сообщение `AssertionError` без информации (рис. 4-1). Вы не знаете, что именно пошло не так, не знаете, какие именно значения проверялись. Вы не знаете ничего.

Пример 4-22. Пример с `assertTrue`

```
assertTrue(documents.size() == 0);
```

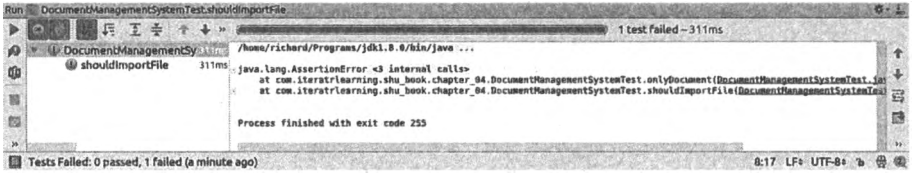

Рис. 4-1. Скриншот с отрицательным тестом `assertTrue`

Самым распространенным утверждением является `assertEquals`, которое берет два значения и проверяет их на равенство. Оно перегружается для проверки примитивных значений. Таким образом, мы можем проверить, что размер документа равен нулю (пример 4-23). Это дает нам чуть более

подробные диагностические результаты (рис. 4-2). Мы знаем, что ожидаемое значение было 0, полученное значение — 1. Однако мы по-прежнему не владеем полным объемом информации.

Пример 4-23. Пример применения `assertEquals`

```
assertEquals(0, documents.size());
```

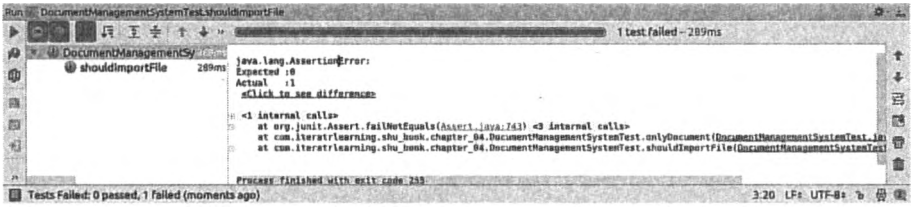

Рис. 4-2. Скриншот с отрицательным тестом `assertEquals`

Лучший способ сделать утверждение о размере коллекции — использовать *матчер*, он позволяет получить более детальную диагностическую информацию. В примере 4-24 показано применение данного способа, а также демонстрируется его результат. Как видно по рис. 4-3, уже становится понятнее, что именно пошло не так.

Пример 4-24. Пример использования `assertThat`

```
assertThat(documents, hasSize(0));
```

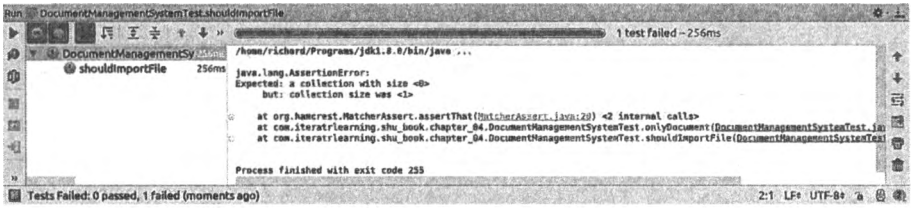

Рис. 4-3. Скриншот с отрицательным результатом теста `assertThat`

Что происходит при использовании `assertThat()` из JUnit. Метод `assertThat()` принимает значение в качестве первого параметра, а матчер — в качестве второго. Матчер инкапсулирует принцип, в котором значение соответствует определенному свойству и его диагностической информации. Матчер `hasSize` статически импортирован из служебного класса `Matchers`,

в котором есть множество различных матчеров. Он проверяет, что размер коллекции соответствует заданному параметру. Эти матчеры пришли из библиотеки `Hamcrest`, широко используемой библиотеки Java, упрощающей тестирование.

Другой способ достижения хорошей диагностики был показан в примере 4-20. В данном случае `assertEquals` должен предоставлять нам диагностическую информацию об ожидаемом и фактическом значении атрибутов. Он не говорит нам об имени атрибута, поэтому эта информация была добавлена в строку сообщения, чтобы лучше распознать отказ.

Тестирование ошибочных ситуаций

Одна из наихудших и наиболее распространенных ошибок в разработке программного обеспечения — тестировать только золотые и прекрасные ситуации в приложении. Ситуации, когда солнце сияет над вами и все идет по плану. На практике много чего может пойти не так.

Если вы не тестируете свое приложение на предмет подобных ситуаций — вы не можете получить программное обеспечение, стабильно работающее в реальных условиях эксплуатации.

Что касается нашей системы управления документами, есть несколько ошибок, которые могут возникнуть в процессе импортирования документов. Мы можем попытаться импортировать файл, который не существует или не читается. Или файл, из которого невозможно извлечь текст.

У `DocumentManagementSystemTest` есть несколько тестов для этих двух сценариев. Они показаны в примере 4-25. В обоих случаях мы пытаемся импортировать файл, который принесет проблемы. Чтобы сделать утверждение об ожидаемом поведении, мы используем атрибут `expected =` из JUnit-аннотации `@Test`. Он позволяет вам сказать: «*Эй, JUnit, послушай, я ожидаю, что данный тест сгенерирует исключение определенного типа*».

Пример 4-25. Тестирование ошибочных ситуаций

```
@Test(expected = FileNotFoundException.class)
public void shouldNotImportMissingFile() throws Exception
{
    system.importFile("gobbledygook.txt");
}
```

```
@Test(expected = UnknownFileTypeException.class)
public void shouldNotImportUnknownFile() throws Exception
{
    system.importFile(RESOURCES + "unknown.txt");
}
```

Возможно, для генерации исключения вам понадобится другое поведение, но в любом случае полезно знать, как сделать утверждение, чтобы сгенерировать исключение.

Константы

Константы представляют собой значения, которые не изменяются. Давайте посмотрим правде в глаза — это одна из немногих хорошо известных концепций программирования. В языке Java не используется ключевое слово `const`, как это делается в C++, обычно для представления констант разработчики создают поля `static field`. Поскольку тесты содержат примеры того, как должна работать та или иная часть вашей программы, в них часто встречаются константы.

Когда речь заходит о константах, имеющих какое-то неочевидное значение, имеет смысл давать им собственные имена, которые можно использовать в тестах.

Мы уже поступали так при работе с тестом `DocumentManagementSystemTest`, в котором есть блок кода, отведенный для объявления констант (пример 4-26).

Пример 4-26. Константы

```
public class DocumentManagementSystemTest
{
    private static final String RESOURCES =
        "src" + File.separator + "test" + File.separator + "resources" + File.
separator;
    private static final String LETTER = RESOURCES + "patient.letter";
    private static final String REPORT = RESOURCES + "patient.report";
    private static final String XRAY = RESOURCES + "xray.jpg";
    private static final String INVOICE = RESOURCES + "patient.invoice";
    private static final String JOE_BLOGGS = "Joe Bloggs";
```

Выводы

- Вы узнали, как создать систему управления документами.
- Вы познакомились с различными компромиссами между разными подходами к реализации.
- Вы познакомились с несколькими основными принципами разработки программного обеспечения.
- Вы узнали о принципе подстановки Лисков как о способе понимания наследования.
- Вы познакомились с ситуациями, когда не стоит применять наследование.

Самостоятельная работа

Если вы хотите расширить и закрепить знания, полученные в данной главе, попробуйте сделать что-нибудь из этого:

- Возьмите существующий пример кода и добавьте в него реализацию импорта рецептов. В рецепте должна быть информация о пациенте, лекарственном препарате, дозировке, схеме и условиях приема препарата. Кроме того, не забудьте написать тест, для проверки импорта рецептов.
- Попробуйте реализовать игру Game of Life Kata[1].

В завершение

Доктор Авадж очень довольна системой управления документами и теперь пользуется ей постоянно. Ее потребности полностью удовлетворены, потому что в процессе разработки вы учитывали ее требования к поведению приложения и важные детали. Мы еще вернемся к данной теме в следующей главе, когда будем знакомиться с TDD.

---

[1] С подробной информацией о задаче вы можете изнакомиться на https://codingdojo.org/kata/GameOfLife — *Прим. авт.*

Движок бизнес-правил

Задача

Ваш бизнес идет хорошо. Ваша организация выросла, и в ней уже тысяча сотрудников. Вы наняли много людей для выполнения различный функций: маркетинг, продажи, администрация, бухгалтерия и т. д. Вы понимаете, что каждая бизнес-задача требует определенных правил, диктующих выполнение тех или иных действий в зависимости от определенных условий. Например, «уведомить отдел продаж, если должность потенциального клиента — генеральный директор». В случае с заказным программным обеспечением вам придется каждый раз обращаться в техническую службу, чтобы реализовать новое правило, а ваши разработчики уже весьма заняты работой над другими проектами. Поэтому, чтобы улучшить взаимодействие между различными подразделениями, вы принимаете решение создать свой движок бизнес-правил, который позволит различным командам совместно разрабатывать программное обеспечение. Это повысит продуктивность и сократит время, затрачиваемое на внедрение новых бизнес-правил, потому что ваши команды смогут взаимодействовать напрямую.

Цель

В данной главе вы впервые познакомитесь с разработкой через тестирование. Вы также узнаете о технике под названием мокинг (имитация), которая поможет улучшить модульные тесты. Затем вы рассмотрите некоторые современные особенности Java: вывод типа локальной переменной и `switch`-выражения (переключатели). И наконец, вы узнаете, как разработать дружественный API при помощи шаблона Builder и принципа разделения интерфейсов.

Если в какой-то момент вы захотите посмотреть код программы этой главы, вы всегда можете найти его в репозитории: `/com/iteratrlearning/shu_book/chapter_05`.

Требования к движку бизнес-правил

Перед тем как начать, давайте подумаем о том, что мы планируем получить в итоге. Вы хотите, чтобы у непрограммистов была возможность добавлять или изменять бизнес-логику их рабочего пространства. Например, работникам отдела маркетинга может понадобиться применить определенную скидку, если клиент запросит определенный продукт. Бухгалтерия заинтересована в том, чтобы система выдавала предупреждение, если расходы сильно возрастают. Это просто примеры того, к чему мы будем стремиться при разработке движка бизнес-правил. Мы планируем создать очень важное программное обеспечение, реализующее одно или несколько бизнес-правил, которые обычно задаются при помощи специального языка заказчика. Движок бизнес-правил может поддерживать различные компоненты:

Факты

Информация, к которой у правил есть доступ.

Действия

Операции, которые должны выполняться.

Условия

Они определяют, когда действия должны запускаться.

Правила

Они определяют саму логику, которую вы хотите реализовать и включают в себя факты, условия и действия.

Ключевое преимущество движка бизнес-правил заключается в том, что он позволяет поддерживать, выполнять и тестировать правила в одном месте без необходимости интеграции с основным приложением.

Есть много готовых Java движков бизнес-правил, например, Drools[1]. Обычно подобные движки соответствуют определенным стандартам вроде DMN (*Decision Model and Notation*) и поставляются с централизованной базой правил, редактором на основе *графического интерфейса пользователя* (GUI) и инструментами визуализации, помогающими обслуживать сложные правила. В этой главе вы разработаете минимально жизнеспособный продукт, и будете работать над улучшением его функциональности и удобства.

Разработка через тестирование

С чего начать? Требования к продукту еще не окончательные и могут в процессе дополняться, поэтому мы начнем со списка основных функциональных возможностей, которыми будут пользоваться ваши сотрудники:

- Добавлять действия.
- Запускать действия.
- Получать базовые отчеты.

Таким образом, мы имеем дело с базовым API, он представлен в примере 5-1. Каждый метод генерирует исключение `UnsupportedOperationException`, показывающее, что его еще нужно реализовать.

Пример 5-1. Базовый API для движка бизнес-правил

```
public class BusinessRuleEngine {

    public void addAction(final Action action) {
        throw new UnsupportedOperationException();
    }

    public int count() {
        throw new UnsupportedOperationException();
    }

    public void run() {
        throw new UnsupportedOperationException();
    }

}
```

[1] Движок Drools доступен по адресу: https://www.drools.org. — *Прим. авт.*

Действие — это просто фрагмент кода, который будет исполняться. Мы могли бы воспользоваться интерфейсом `Runnable`, но внедрение отдельного интерфейса `Action` больше перекликается с доменом. Интерфейс `Action` позволит движку бизнес-правил как бы отделиться от конкретных действий. Так как интерфейс `Action` только объявляет один абстрактный метод, мы можем считать, что это функциональный интерфейс (пример 5-2).

Пример 5-2. Интерфейс `Action`

```
@FunctionalInterface
public interface Action {
    void execute();
}
```

Что дальше? Настало время приступить к реализации и написать какой-то код. В этом вам поможет знакомство с подходом *разработки через тестирование* (TDD — test-driven development). Философия TDD предполагает разработку тестов, которые затем приводят к реализации кода программы. Другими словами, сначала вы пишете тесты, а потом основную реализацию. Данный подход может показаться несколько необычным, ведь он, по сути, противоположен тому, чем мы занимались ранее: сначала писали код в соответствии с требованиями, а только потом тестировали его. Теперь же мы сконцентрируемся больше на тестах.

Зачем нужен TDD?

Почему вы должны прибегнуть к разработке через тестирование? Дело в том, что у данного подхода есть ряд полезных преимуществ:

- Написание теста помогает сфокусироваться и конкретизировать требования за счет корректной реализации одного элемента за один раз.
- Это возможность найти наилучший способ организации кода. Например, если первоначально вы пишете тест, это вынуждает вас хорошенько подумать о реализации публичных интерфейсов вашей программы.
- По ходу последовательного изучения требований вы создаете комплексный набор тестов, что повышает соответствие кода требованиям к программе и ведет к снижению количества багов.
- Вы не создаете код, который вам не нужен, потому что пишете только код, который проходит тестирование.

Цикл TDD

Подход TDD условно состоит из следующих этапов, представленных на рис. 5-1:

1. Написание теста, который дает отрицательный результат.
2. Запуск всех тестов.
3. Выполнение рабочей реализации.
4. Запуск всех тестов.

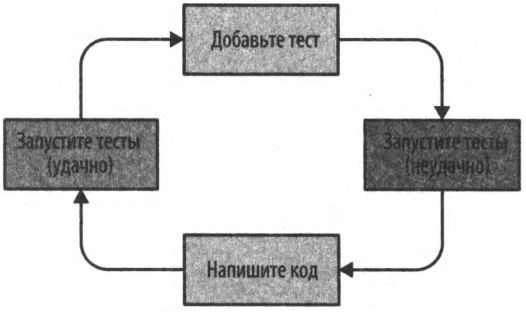

Рис. 5-1. Цикл TDD

На практике в этот процесс также входит непрерывная работа над *усовершенствованием* кода. В противном случае можно получить необслуживаемый код. На данный момент у вас есть некий набор тестов, на который вы можете положиться при внесении изменений в программу. На рис. 5-2 продемонстрирован улучшенный цикл TDD.

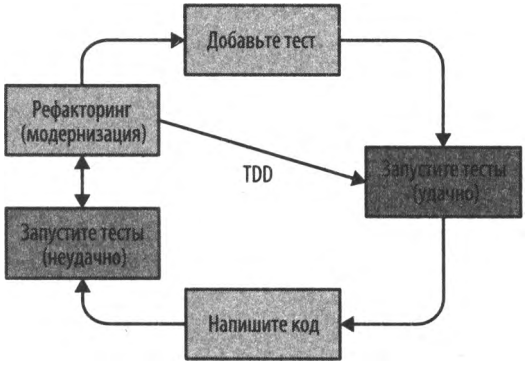

Рис. 5-2. TDD с рефакторингом

Итак, давайте начнем с написания первых тестов, проверяющих, что `addActions` и `count` работают корректно (пример 5-3).

Пример 5-3. Базовый тест для движка бизнес-правил

```
@Test
void shouldHaveNoRulesInitially() {
    final BusinessRuleEngine businessRuleEngine = new BusinessRuleEngine();

    assertEquals(0, businessRuleEngine.count());
}

@Test
void shouldAddTwoActions() {
    final BusinessRuleEngine businessRuleEngine = new BusinessRuleEngine();

    businessRuleEngine.addAction(() -> {});
    businessRuleEngine.addAction(() -> {});

    assertEquals(2, businessRuleEngine.count());
}
```

При запуске тестов вы увидите, что они завершаются неудачно, выбрасывая исключение `UnsupportedOperationException`, как показано на рис. 5-3.

Рис. 5-3. Неудачные тесты

Все тесты неудачные, но это хорошо. Мы получаем воспроизводимый набор тестов, который поможет при реализации кода. Сейчас давайте займемся самой реализацией (пример 5-4).

Пример 5-4. Базовая реализация движка бизнес-правил

```
public class BusinessRuleEngine {

    private final List<Action> actions;
```

```
    public BusinessRuleEngine() {
        this.actions = new ArrayList<>();
    }

    public void addAction(final Action action) {
        this.actions.add(action);
    }

    public int count() {
        return this.actions.size();
    }

    public void run(){
        throw new UnsupportedOperationException();
    }
}
```

Теперь вы можете запустить тесты — и да, они проходят успешно. Однако мы упускаем один важный момент. Как нам написать тест для метода `run`? Как ни странно, `run()` не возвращает никаких значений. Нам нужно прибегнуть к новой методике, которая называется мокинг (имитация), позволяющей проверить функционирование метода `run()`.

Мокинг

Мокинг — это методика, которая позволит вам проверить, что каждое действие, добавленное в движок бизнес-правил, будет выполнено при запуске метода `run()`. В данный момент это сделать достаточно сложно, потому что метод `run()` в `BusinessRuleEngine` и метод `perform()` в `Action` не возвращают значений. У нас нет возможности создать утверждение! Более подробно мокинг рассматривается в главе 6, но кратко мы познакомимся с ним сейчас, чтобы немного продвинуться в написании теста. Для этого необходимо воспользоваться Mockito — популярной мокинг-библиотекой Java. Для начала вы можете сделать две вещи:

1. Создать мок.
2. Убедиться, что метод вызван.

   Но прежде нужно импортировать библиотеку:

   ```
   import static org.mockito.Mockito.*;
   ```

Теперь вы можете пользоваться методами `mock()` и `verify()`. Статический метод `mock()` позволяет создать мок-объект, который потом можно проверить на определенное поведение. Метод `verify()` позволяет сделать утверждения о том, что вызван определенный метод. В примере 5-5 показано, как это выглядит.

Пример 5-5. Имитация и проверка взаимодействия с объектом `Action`

```
@Test
void shouldExecuteOneAction() {
        final BusinessRuleEngine businessRuleEngine = new BusinessRuleEngine();
        final Action mockAction = mock(Action.class);

        businessRuleEngine.addAction(mockAction);
        businessRuleEngine.run();

        verify(mockAction).perform();
}
```

Модульный тест создает мок-объект для `Action`. Это осуществляется путем передачи класса в качестве аргумента мок-методу. Затем идет условная часть теста, *в которой* вы вызываете интересующее вас действие (поведение). Здесь вы добавляете действие и запускаете метод `run()`. В конце идет *результирующая* часть модульных тестов, в которой устанавливаются утверждения. В нашем случае проверяется, что метод `perform()` был вызван для объекта `Action`. Если вы запустите данный тест — он, как и ожидается, будет провален и выдаст исключение `UnsupportedOperationException`. Что, если тело метода `run()` окажется пустым? Вы получите новое исключение с трассировкой:

```
Wanted but not invoked:
action.perform();
-> at BusinessRuleEngineTest.shouldExecuteOneAction(BusinessRuleEngineTest.java: 35)
Actually, there were zero interactions with this mock.
```

Данная ошибка создана библиотекой Mockito и говорит вам о том, что метод `perform()` не был вызван. Настало время написать нормальную реализацию метода `run()`, как показано в примере 5-6.

Пример 5-6. Реализация метода `run()`

```
public void run() {
    this.actions.forEach(Action::perform);
}
```

После повторного запуска тестов вы увидите, что они успешно выполняются. Mockito удалось проверить, что при запуске движка бизнес-правил должен вызываться метод `perform()` для объекта `Action`. Mockito позволяет вам устанавливать сложную логику тестирования, такую как проверка количества вызовов метода, в том числе с определенными аргументами. Более детально вы изучите это в главе 6.

Добавление условий

Вы должны признать, что пока ваш движок бизнес-правил имеет довольно ограниченные возможности. Вы можете объявлять только простые действия. Однако на практике пользователям часто будет необходимо создавать правила, основанные на определенных условиях, которые, в свою очередь, зависят от определенных фактов. Например, уведомлять отдел продаж *только* в том случае, *если* должность заказчика — *генеральный директор*.

Моделирование состояния

Вы можете начать с написания кода, который добавляет действия и связан с некоторой локальной переменной через анонимный класс (как это показано в примере 5-7), или при помощи лямбда-выражения (пример 5-8).

Пример 5-7. Добавление действия через анонимный класс

```
// this object could be created from a form
final Customer customer = new Customer("Mark", "CEO");

businessRuleEngine.addAction(new Action() {

    @Override
    public void perform() {
        if ("CEO".equals(customer.getJobTitle())) {
            Mailer.sendEmail("sales@company.com", "Relevant customer: " + customer);
        }
    }
});
```

Пример 5-8. Добавление действия через лямбда-выражение

```
// this object could be created from a form
final Customer customer = new Customer("Mark", "CEO");

businessRuleEngine.addAction(() -> {
    if ("CEO".equals(customer.getJobTitle())) {
        Mailer.sendEmail("sales@company.com", "Relevant customer: " + customer);
    }
});
```

Стоит отметить, что данный подход не совсем удобен по нескольким причинам:

1. Как тестировать действие? Это не независимый функциональный элемент. У него есть сложная зависимость от пользовательского объекта.
2. Пользовательский объект не сгруппирован с самим действием. Это определенного рода внешнее состояние, которое находится «где-то там», и приводит к запутыванию ответственности.

Так что же нам нужно? Нам нужно инкапсулировать это состояние. Давайте смоделируем необходимые требования при помощи нового класса `Facts`, который будет представлять собой состояние, доступное для движка бизнес-правил, а также при помощи обновленного интерфейса `Action`, который может работать с фактами. Обновленный модульный тест представлен в примере 5-9. Он проверяет, что при запуске движка бизнес-правил, указанное действие действительно вызывается с объектом `Facts`, переданным в качестве аргумента.

Пример 5-9. Тестирование действия с фактами

```
@Test
public void shouldPerformAnActionWithFacts() {
    final Action mockAction = mock(Action.class);
    final Facts mockFacts = mock(Facts.class);
    final BusinessRuleEngine businessRuleEngine = new BusinessRuleEngine(mockedFacts);

    businessRuleEngine.addAction(mockAction);
    businessRuleEngine.run();

    verify(mockAction).perform(mockFacts);
}
```

Исходя из философии TDD, изначально наш тест должен быть провальным. Всегда нужно начинать с запуска тестов и убеждаться, что они дают отрицательный результат. Хотя вы можете написать тест, который случайно пройдет. Чтобы тест выполнялся успешно, вам нужно обновить API и код реализации. Для начала необходимо ввести класс `Facts`, который позволит вам хранить факты, представленные в виде некоторого ключа и соответствующего значения. Преимущество введения отдельного класса `Facts` для моделирования состояния заключается в том, что вы получаете возможность управлять операциями, доступными пользователям за счет публичного API, а также применять модульные тесты для контроля поведения класса. Пока что класс `Facts` будет поддерживать ключи и значения типа `String`. Код класса `Facts` представлен в примере 5-10. Мы выбрали имена `getFact` и `addFact` потому что они лучше представляют связь с доменом (работа с фактами), чем `getValue` и `setValue`.

Пример 5-10. Класс `Facts`

```
public class Facts {

    private final Map<String, String> facts = new HashMap<>();

    public String getFact(final String name) {
        return this.facts.get(name);
    }

    public void addFact(final String name, final String value) {
        this.facts.put(name, value);
    }
}
```

Теперь вам необходимо переработать интерфейс `Action`, чтобы метод `perform()` мог принимать объекты `Facts` в качестве аргументов. Таким образом становится ясно, что факты доступны в контексте одного действия (пример 5-11).

Пример 5-11. Интерфейс `Action`, который принимает факты

```
@FunctionalInterface
public interface Action {
    void perform(Facts facts);
}
```

Наконец, вы можете обновить класс BusinessRuleEngine, чтобы задействовать факты и обновленный метод perform() (пример 5-12).

Пример 5-12. BusinessRuleEngin с фактами

```
public class BusinessRuleEngine {

    private final List<Action> actions;

    private final Facts facts;

    public BusinessRuleEngine(final Facts facts) {
        this.facts = facts;
        this.actions = new ArrayList<>();
    }

    public void addAction(final Action action) {
        this.actions.add(action);
    }

    public int count() {
        return this.actions.size();
    }

    public void run() {
        this.actions.forEach(action -> action.perform(facts));
    }
}
```

Теперь, когда объекты Facts доступны действиям, вы можете конкретизировать избирательную логику в вашей программе, которая учитывает объекты Facts. Это демонстрируется в примере 5-13.

Пример 5-13. Действие с задействованными фактами

```
businessRuleEngine.addAction(facts -> {
    final String jobTitle = facts.getFact("jobTitle");
    if ("CEO".equals(jobTitle)) {
        final String name = facts.getFact("name");
        Mailer.sendEmail("sales@company.com", "Relevant customer: " + name);
    }
});
```

Давайте рассмотрим еще пару примеров, которые позволят нам узнать кое-что новое о Java. Речь идет о следующих особенностях:

- Вывод типа локальной переменной.
- `Switch`-выражения.

Вывод типа локальной переменной

В Java 10 появилась возможность вывода типа локальной переменной. Идея в том, что компилятор может определять статические типы вместо вас, так что вам не нужно указывать их. Вы уже сталкивались с примером вывода типа ранее, в примере 5-10, когда написали:

```
Map<String, String> facts = new HashMap<>();
```

вместо

```
Map<String, String> facts = new HashMap<String, String>();
```

Эта особенность была представлена в Java 7 и называлась «*бриллиантовый оператор*» (diamond operator). По сути, вы можете пропустить параметры исходных типов (в нашей ситуации `String, String`) в выражении, если они определяются в контексте. В коде выше левая часть присвоения указывает, что ключи и значения `Map` должны быть `String`.

Начиная с Java 10, вывод типа переменной расширен для работы с локальными переменными. Например, код в примере 5-14 может быть написан с использованием ключевого слова `var`. Вывод типа локальной переменной показан в примере 5-15.

Пример 5-14. Объявление локальной переменной с указанием типов

```
Facts env = new Facts();
BusinessRuleEngine businessRuleEngine = new BusinessRuleEngine(env);
```

Пример 5-15. Вывод типа локальной переменной

```
var env = new Facts();
var businessRuleEngine = new BusinessRuleEngine(env);
```

За счет использования ключевого слова `var` в примере 5-15, переменная `env` все еще статического типа `Facts`, а переменная `businessRuleEngine` по-прежнему статического типа `BusinessRuleEngine`.

Переменная, объявленная с помощью ключевого слова `var`, не является `final`. Например, код:

```
final Facts env = new Facts();
```

не эквивалентен коду:

```
var env = new Facts();
```

После объявления переменной `env` при помощи `var` вы все еще можете присвоить ей другое значение. Поэтому вам нужно указать ключевое слово `final` в начале объявления переменной `env`, как показано ниже:

```
final var env = new Facts()
```

В остальных главах мы используем ключевое слово `var` без слова `final` для краткости, так как это выглядит более лаконично. Когда мы явно объявляем тип переменной, то используем слово `final`.

Выведение типа позволяет несколько сократить время, затрачиваемое на написание Java-кода. Однако стоит ли пользоваться этой возможностью постоянно? Нужно помнить, что большую часть времени разработчики читают код, а не пишут. Другими словами, вы должны отдавать предпочтение упрощению чтения, а не написания кода. Степень, в которой `var` помогает добиться этого, всегда будет понятием субъективным. Поскольку в первую очередь необходимо думать о том, что может помочь вашим коллегам при чтении вашего кода, не лишним будет поинтересоваться их мнением. Если им нравится читать код с `var`, то, конечно, смело вперед. Иначе — нет. Например, в данном случае мы можем преобразовать код из примера 5-13 с использованием вывода типа локальной переменной, и получить код, представленный в примере 5-16.

Пример 5-16. Действие с задействованными фактами и выводом типа локальной переменной

```
businessRuleEngine.addAction(facts -> {
    var jobTitle = facts.getFact("jobTitle");
    if ("CEO".equals(jobTitle)) {
        var name = facts.getFact("name");
        Mailer.sendEmail("sales@company.com", "Relevant customer: " + name);
    }
});
```

Switch-выражения

Недавно вы задавали действия только с одним условием для обработки. Этого явно недостаточно. Например, вы работаете с отделом продаж. В свою

систему по работе с клиентами (Customer Relationship Management — CRM) они могут записывать различные сделки, имеющие разные статусы (этапы) и разные суммы. Статус сделки может быть представлен в виде перечисления `Stage` со значениями, включающими `LEAD`, `INTERESTED`, `EVALUATING`, `CLOSED` (ведется, есть заинтересованность, на согласовании, закрыта) — пример 5-17.

Пример 5-17. Перечисление, представляющее различные стадии сделки

```
public enum Stage {
    LEAD, INTERESTED, EVALUATING, CLOSED
}
```

В зависимости от стадии сделки вы можете применить правило, дающее вам возможность ее выиграть. Следовательно, вы можете помочь отделу продаж делать прогнозы. Предположим, у определенной команды ведущая сделка имеет 20%-ную вероятность завершения, в таком случае сделка стоимостью 1000 долларов имеет прогноз в 200 долларов. Давайте создадим событие, чтобы смоделировать эти правила и вычислить прогнозируемую сумму конкретной сделки (как показано в примере 5-18).

Пример 5-18. Правило для вычисления прогнозируемой суммы конкретной сделки

```
businessRuleEngine.addAction(facts -> {
    var forecastedAmount = 0.0;
    var dealStage = Stage.valueOf(facts.getFact("stage"));
    var amount = Double.parseDouble(facts.getFact("amount"));
    if(dealStage == Stage.LEAD){
        forecastedAmount = amount * 0.2;
    } else if (dealStage == Stage.EVALUATING) {
        forecastedAmount = amount * 0.5;
    } else if(dealStage == Stage.INTERESTED) {
        forecastedAmount = amount * 0.8;
    } else if(dealStage == Stage.CLOSED) {
        forecastedAmount = amount;
    }
    facts.addFact("forecastedAmount", String.valueOf(forecastedAmount));
});
```

Код из примера 5-18 по существу определяет значение для каждого доступного элемента перечисления. Предпочтительнее в данном случае использовать оператор `switch`, поскольку он более лаконичен (пример 5-19).

Пример 5-19. Правило для вычисления прогнозируемой суммы сделки на основе оператора `switch`

```
switch (dealStage) {
    case LEAD:
        forecastedAmount = amount * 0.2;
        break;
    case EVALUATING:
        forecastedAmount = amount * 0.5;
        break;
    case INTERESTED:
        forecastedAmount = amount * 0.8;
        break;
    case CLOSED:
        forecastedAmount = amount;
        break;
}
```

Обратите внимание на операторы `break` в примере 5-19. Оператор `break` нужен для того, чтобы остановить выполнение следующего блока конструкции `switch`. Если вы случайно забудете поставить `break`, код продолжит компилироваться, и вы получите *неадекватное* поведение программы. Другими словами, последующие блоки будут тоже выполняться, что приведет к багу. Начиная с Java 12 (при использовании режима предпросмотра), вы можете при помощи различного синтаксиса `switch` избежать большого количества `break` и неадекватного поведения кода. Теперь `switch` может использоваться как выражение (пример 5-20).

Пример 5-20. Выражение `switch` *без «сквозного» поведения*

```
var forecastedAmount = amount * switch (dealStage) {
    case LEAD -> 0.2;
    case EVALUATING -> 0.5;
    case INTERESTED -> 0.8;
    case CLOSED -> 1;
}
```

Другим преимуществом «доработанного» `switch` является улучшенная читаемость и *исчерпываемость*. Это означает, что при использовании `switch` с перечислением компилятор Java проверяет, что для всех значений перечисления существует соответствующая метка (сценарий) `switch`. Например,

если вы забудете обработать вариант CLOSED, компилятор Java выдаст вам следующую ошибку:

```
error: the switch expression does not cover all possible input values.
```

Вы можете преобразовать все действие при помощи оператора switch, как это показано в примере 5-21.

Пример 5-21. Правило для вычисления прогнозируемой суммы сделки

```
businessRuleEngine.addAction(facts -> {
    var dealStage = Stage.valueOf(facts.getFact("stage"));
    var amount = Double.parseDouble(facts.getFact("amount"));
    var forecastedAmount = amount * switch (dealStage) {
        case LEAD -> 0.2;
        case EVALUATING -> 0.5;
        case INTERESTED -> 0.8;
        case CLOSED -> 1;
    }
    facts.addFact("forecastedAmount", String.valueOf(forecastedAmount));
});
```

Принцип разделения интерфейса

Сейчас предлагаем разработать *инспектирующий инструмент*, позволяющий пользователям движка бизнес-правил инспектировать статус возможных действий и состояний. Допустим, мы хотим оценивать каждое действие и связанное с ним состояние, чтобы зафиксировать эту информацию, собственно, без выполнения действия. Как нам быть? Текущая версия интерфейса Action не отвечает данной задаче, потому что она не разделяет выполняемый код и условие, этот код запускающее. На данный момент нет возможности разделить исполняемый код и условие. Чтобы ее получить, нам придется ввести доработанный интерфейс Action со встроенным функционалом оценки условий. Например, мы можем создать интерфейс ConditionalAction, в котором есть новый метод evaluate(), как показано в примере 5-22.

Пример 5-22. Интерфейс ConditionalAction

```
public interface ConditionalAction {
    boolean evaluate(Facts facts);
    void perform(Facts facts);
}
```

Теперь мы в состоянии реализовать базовую версию класса Inspector, принимающего список объектов ConditionalAction и оценивающего их на основе некоторых фактов (пример 5-23). Inspector возвращает список отчетов, в которых находятся факты, условные действия и результат. Реализация класса Report показана в примере 5-24.

Пример 5-23. Инспектор условий

```
public class Inspector {

    private final List<ConditionalAction> conditionalActionList;

    public Inspector(final ConditionalAction...conditionalActions) {
        this.conditionalActionList = Arrays.asList(conditionalActions);
    }

    public List<Report> inspect(final Facts facts) {
        final List<Report> reportList = new ArrayList<>();
        for (ConditionalAction conditionalAction : conditionalActionList) {
            final boolean conditionResult = conditionalAction.evaluate(facts);
            reportList.add(new Report(facts, conditionalAction, conditionResult));
        }
        return reportList;
    }
}
```

Пример 5-24. Класс Report

```
public class Report {

    private final ConditionalAction conditionalAction;
    private final Facts facts;
    private final boolean isPositive;

    public Report(final Facts facts,
                  final ConditionalAction conditionalAction,
                  final boolean isPositive) {
        this.facts = facts;
        this.conditionalAction = conditionalAction;
        this.isPositive = isPositive;
    }

    public ConditionalAction getConditionalAction() {
        return conditionalAction;
    }
```

```
    public Facts getFacts() {
        return facts;
    }

    public boolean isPositive() {
        return isPositive;
    }

    @Override
    public String toString() {
        return "Report{" +
                "conditionalAction=" + conditionalAction +
                ", facts=" + facts +
                ", result=" + isPositive +
                '}';
    }
}
```

Как нам поступить с тестированием `Inspector`? Начнем с написания простого модульного теста, показанного в примере 5-25. Данный тест выявляет фундаментальную проблему нашей разработки. По факту интерфейс `ConditionalAction` нарушает *принцип разделения интерфейса* (ISP — Interface Segregation Principle).

Пример 5-25. Пояснение противоречия ISP

```
public class InspectorTest {

    @Test
    public void inspectOneConditionEvaluatesTrue() {

        final Facts facts = new Facts();
        facts.setFact("jobTitle", "CEO");
        final ConditionalAction conditionalAction = new JobTitleCondition();
        final Inspector inspector = new Inspector(conditionalAction);

        final List<Report> reportList = inspector.inspect(facts);

        assertEquals(1, reportList.size());
        assertEquals(true, reportList.get(0).isPositive());
    }

    private static class JobTitleCondition implements ConditionalAction {
```

```
        @Override
        public void perform(Facts facts) {
            throw new UnsupportedOperationException();
        }

        @Override
        public boolean evaluate(Facts facts) {
            return "CEO".equals(facts.getFact("jobTitle"));
        }
    }
}
```

Что такое принцип разделения интерфейса? Вы можете заметить, что реализация метода `perform` пока пуста. На самом деле он генерирует исключение `UnsupportedOperationException`. В данной ситуации вы привязаны к интерфейсу (`ConditionalAction`), который делает больше, чем вам нужно. В этом случае вам нужен способ моделирования условия — что-то, что проверяет на истинность. Тем не менее вы насильно зависите от метода `perform`, потому что он является частью интерфейса.

Основная идея принципа разделения интерфейса заключается в том, что класс не должен зависеть от метода, которым он не пользуется, потому что это приводит к ненужной связанности. В главе 2 вы познакомились с другим принципом — *принципом единой ответственности*, говорящим о необходимости поддержания высокой связности. Принцип единой ответственности — это ключевой принцип разработки, он диктует нам, что класс несет ответственность за определенный функционал и есть только одна причина для его изменения.

Несмотря на то что принцип ISP, возможно, звучит похоже, он предлагает несколько иной взгляд. Он фокусируется на пользователе интерфейса, а не его наполнении. Другими словами, если интерфейс получается очень большим, скорее всего, его пользователь видит много возможностей, которые его не интересуют. Это приводит к нежелательному повышению связанности.

Чтобы разработать решение, отвечающее принципу разделения интерфейса, мы собираемся разделить наши задачи, используя более маленькие интерфейсы, которые можно проверять раздельно. Эта идея, по сути, способствует повышению связности. Разделение интерфейсов также позволят применять имена, более приближенные к домену, такие как `Condition` и `Action`. Данный вопрос мы раскроем в следующем разделе.

Разработка текучего интерфейса (Fluent API)

Недавно мы реализовали возможность добавления действия со сложными условиями. Данные условия были созданы при помощи улучшенного оператора `switch`. Однако синтаксис задания условий достаточно сложен для простых сотрудников. Мы бы хотели упростить им жизнь, предоставив возможность задавать правила (условие и действие) способом, соответствующим их роду деятельности. В этом разделе вы познакомитесь с шаблоном Builder и узнаете, как разработать свой собственный Fluent API (текучий интерфейс), чтобы решить данную проблему.

Что такое Fluent API?

Fluent API (текучий интерфейс) — это API, разработанный непосредственно для определенного домена, что позволяет решать конкретные задачи более интуитивно. Он также охватывает идею цепного (последовательного) вызова методов для реализации сложных операций. Существует несколько известных Fluent API, с которыми вы можете быть знакомы:

- Java Streams API[1] позволяет устанавливать запросы обработки данных способом, близким к проблеме, которую вы решаете.
- Spring Integration[2] предлагает Java API устанавливать шаблоны интеграции с предприятием при помощи словаря, близкого к доменным шаблонам интеграции.
- jQQQ[3] предлагает библиотеку для взаимодействия с различными базами данных при помощи интуитивного API.

Моделирование домена

Итак, что именно мы стремимся облегчить нашим пользователям? Мы хотим помочь им задавать простую комбинацию «если соблюдаются такие-то условия», «то сделать то-то» в виде правила. В данном домене есть несколько элементов.

Условие

Условие, применяющееся к определенным фактам, для их проверки на истинность.

---

[1] Подробнее на https://oreil.ly/549wN. — *Прим. авт.*
[2] Подробнее на https://oreil.ly/rMIMD. — *Прим. авт.*
[3] Подробнее на https://www.jooq.org. — *Прим. авт.*

Действие

Определенный набор операций, подлежащих выполнению.

Правило

Условие и действие вместе. Действие выполняется, только если условие истинно.

Теперь, когда мы проговорили основные элементы, пора перенести их в Java. Для начала давайте объявим интерфейс `Condition` и повторно используем уже существующий интерфейс `Action`, как показано в примере 5-26. Заметьте, мы также могли бы воспользоваться интерфейсом `java.util.function.Predicate`, который предоставляется начиная с Java 8, однако имя `Condition` гораздо лучше подходит нашему домену.

Еще раз подчеркнем, что присвоение имен — очень важный момент в программировании. Правильное имя помогает в понимании проблемы, которую решает код. Зачастую имена более важны, чем сам интерфейс (в смысле его параметров и возвращаемых значений), поскольку передают контекстную информацию человеку, читающему код.

Пример 5-26. Интерфейс `Condition`

```
@FunctionalInterface
public interface Condition {
    boolean evaluate(Facts facts);
}
```

Остается вопрос, как смоделировать правило? Мы можем объявить интерфейс `Rule` с действием `perform()`. Это позволит нам создавать различные реализации `Rule`. Подходящая базовая реализация данного интерфейса — это класс `DefaultRule`, работающий с объектами `Action` и `Condition` с соответствующей логикой для реализации правила (пример 5-27).

Пример 5-27. Моделирование правила

```
@FunctionalInterface
interface Rule {
    void perform(Facts facts);
}
```

```
public class DefaultRule implements Rule {

    private final Condition condition;
    private final Action action;

    public Rule(final Condition condition, final Action action) {
        this.condition = condition;
        this.action = action;
    }

    public void perform(final Facts facts) {
        if(condition.evaluate(facts)){
            action.execute(facts);
        }
    }
}
```

Как нам создавать новые правила при помощи всех этих элементов? Ответ — в примере 5-28.

Пример 5-28. Создание правила

```
final Condition condition = (Facts facts) -> "CEO".equals(facts.getFact("jobTi
tle"));
final Action action = (Facts facts) -> {
      var name = facts.getFact("name");
      Mailer.sendEmail("sales@company.com", "Relevant customer!!!: " + name);
};

final Rule rule = new DefaultRule(condition, action);
```

Шаблон Builder

Несмотря на то что в коде используются имена, довольно близкие к домену (`Condition`, `Action`, `Rule`), его все еще можно назвать довольно «ручным». Пользователю нужно создавать экземпляры различных объектов и собирать все вместе. Предлагаем познакомиться с *шаблоном Builder*, который поможет нам улучшить процесс создания объекта `Rule` с соответствующим условием и действием. Целью применения данного шаблона является упрощение создания объектов. Шаблон Builder непосредственно деконструирует параметры конструктора, а вместо этого предоставляет метод для задания каждого

из параметров. Преимущество такого подхода состоит в том, что он позволяет нам объявлять методы с именами, соответствующими домену. Например, в нашем случае мы хотели бы использовать слова «когда» (when) и «тогда» (then). В коде в примере 5-29 показано, как настроить шаблон Builder для создания объекта DefaultRule. Мы вводим метод when(), который устанавливает условие. Метод when() возвращает this (т. е. текущий экземпляр), что позволяет нам запустить следующие методы по цепочке. Кроме того, мы вводим метод then(), который устанавливает действие. Метод then() тоже возвращает this, что опять же дает нам возможность запускать методы по цепочке. Наконец, метод createRule() отвечает за создание объекта DefaultRule().

Пример 5-29. Шаблон Builder для Rule

```
public class RuleBuilder {
    private Condition condition;
    private Action action;

    public RuleBuilder when(final Condition condition) {
        this.condition = condition;
        return this;
    }

    public RuleBuilder then(final Action action) {
        this.action = action;
        return this;
    }

    public Rule createRule() {
        return new DefaultRule(condition, action);
    }
}
```

Используя этот новый класс, вы можете создать RuleBuilder и конфигурировать Rule при помощи методов when(), then() и createRule(), как показано в примере 5-30. Идея «сцепления» методов является ключевым аспектом разработки текучего API (Fluent API).

Пример 5-30. Использование RuleBuilder

```
Rule rule = new RuleBuilder()
        .when(facts -> "CEO".equals(facts.getFact("jobTitle")))
        .then(facts -> {
            var name = facts.getFact("name");
```

```
            Mailer.sendEmail("sales@company.com", "Relevant customer: " + name);
        })
        .createRule();
```

Данный код больше похож на запрос и использует домен: представление правила, `then()` и `when()` в виде встроенных конструкций. Однако этого не достаточно, поскольку у вас все еще остается две неуклюжих конструкции, с которыми придется работать пользователю вашего API:

- Создание «пустого» `RuleBuilder`.
- Вызов метода `createRule()`.

Мы можем исправить это при помощи доработанного API. Есть несколько возможных улучшений:

- Сделать конструктор приватным, чтобы он не мог быть напрямую вызван пользователем. Это значит, что нам нужно будет работать с разными точками входа в API.
- Сделать метод `when()` статичным, чтобы он вызывался напрямую и, по сути, подключал вызов старого конструктора. Кроме того, использование статического метода проясняет вопрос о том, какой метод взять для настройки объектов `Rule`.
- Метод `then()` будет отвечать за создание объекта `DefaultRule`.

В примере 5-31 показан улучшенный `RuleBuilder`.

Пример 5-31. Улучшенный `RuleBuilder`

```
public class RuleBuilder {
    private final Condition condition;

    private RuleBuilder(final Condition condition) {
        this.condition = condition;
    }

    public static RuleBuilder when(final Condition condition) {
        return new RuleBuilder(condition);
    }

    public Rule then(final Action action) {
        return new DefaultRule(condition, action);
    }
}
```

Теперь вы можете легко создавать правила при помощи запуска метода `RuleBuilder.when()`, за которым следует метод `then()`, как показано в примере 5-32.

Пример 5-32. Использование улучшенного `RuleBuilder`

```
final Rule ruleSendEmailToSalesWhenCEO = RuleBuilder
        .when(facts -> "CEO".equals(facts.getFact("jobTitle")))
        .then(facts -> {
            var name = facts.getFact("name");
            Mailer.sendEmail("sales@company.com", "Relevant customer!!!: " + name);
        });
```

И наконец, после модернизации `RuleBuilder` мы можем переработать движок бизнес-правил, чтобы он поддерживал правила вместо просто действий (пример 5-33).

Пример 5-33. Обновленный движок бизнес-правил

```
public class BusinessRuleEngine {

    private final List<Rule> rules;
    private final Facts facts;

    public BusinessRuleEngine(final Facts facts) {
        this.facts = facts;
        this.rules = new ArrayList<>();
    }

    public void addRule(final Rule rule) {
        this.rules.add(rule);
    }

    public void run() {
        this.rules.forEach(rule -> rule.perform(facts));
    }

}
```

Выводы

- Философия разработки через тестирование предполагает начинать с написания тестов, которые потом помогают при переходе к реализации программы.

- Мокинг позволяет писать модульные тесты, которые делают утверждение о том, что определенные действия были запущены.
- Java поддерживает вывод типа локальной переменной и `switch`-выражения.
- Шаблон Builder способствует разработке дружелюбного API для создания экземпляров сложных объектов.
- Принцип разделения интерфейса способствует повышению связности за счет уменьшения зависимости от ненужных методов. Это достигается путем разбивки крупных интерфейсов на более маленькие связные интерфейсы, благодаря чему пользователи не видят ничего лишнего — того, что им не нужно.

Самостоятельная работа

Если вы хотите закрепить или расширить знания, полученные в этой главе, вы можете попробовать сделать что-нибудь из этого:

- Улучшите `Rule` и `RuleBuilder` для поддержки имен и описания.
- Улучшите класс `Facts`, чтобы факты можно было загружать из JSON-файлов.
- Улучшите движок бизнес-правил для реализации поддержки множественных условий.
- Улучшите движок бизнес-правил для поддержки правил с различными приоритетами.

В завершение

Ваш бизнес быстро растет. Компания успешно адаптировала движок бизнес-правил под свой рабочий процесс. Теперь ваши мысли заняты другой идеей, вы хотите применить свои навыки в разработке чего-то нового, что будет полезно всему миру, а не только вашей компании. Настало время перейти к следующей главе и познакомиться с проектом Twootr!

Twootr

Задача

Джо оказался взволнованным молодым человеком. Он торопился рассказать нам о своей новой стартап-идее. Джо хочет помочь людям общаться лучше и быстрее. Ему нравилось вести блоги, и он задумался о том, как привлечь больше людей с меньшими затратами. Он назвал это микроблогингом. Его идея заключается в том, чтобы сократить длину сообщения до 140 символов. Так, по его мнению, люди будут делать публикации чаще, размещая много коротеньких постов вместо длинных сообщений.

Мы спросили Джо, на самом ли деле он считает, что сокращение количества символов побудит людей публиковать короткие, лаконичные сообщения, которые, по сути, ничего не значат. Он ответил: «Живем только раз!» Мы спросили Джо, как именно он собирается зарабатывать деньги. Он сказал: «Живем только раз!» Мы спросили Джо, как он собирается назвать свой продукт. Он ответил: «Twootr!» Мы подумали, что это звучит довольно круто и оригинально и решили помочь ему в реализации его идеи.

Цель

Данная глава поможет вам увидеть полную картину процесса сборки программного обеспечения в единое целое. Большинство предыдущих приложений, описанных в этой книге, были небольшими и работали из командной строки. Twootr — это серверное Java-приложение, соответственно, оно уже ближе к тем приложениям, которые создает большинство Java-разработчиков.

В этой главе у вас будет возможность получить знания по следующим вопросам:

- Как получить целостное описание и разбить его на отдельные архитектурные решения.
- Как использовать тесты-дублеры и тестовые взаимодействия с различными элементами вашего кода.
- Как научиться пользоваться тактикой «от большого к малому» и двигаться от общих требований к самому ядру вашего приложения.

В этой главе мы будем говорить не только о конечном результате нашей работы, но и о том, как к нему прийти. Мы приведем несколько примеров, наглядно демонстрирующих, как определенные методы постепенно развиваются по мере разработки проекта в соответствии со списком реализованных возможностей. Это даст вам реальное представление о том, как могут развиваться программные проекты.

Требования к Twootr

Все предыдущие приложения, описанные в книге, были ориентированы на бизнес и предназначены для работы с данными и документами. Twootr — совсем другое дело, это пользовательское приложение. Когда мы разговаривали с Джо о требованиях к его системе, он внес ряд важных уточнений. Каждый пользовательский микроблог должен называться twoot (твут), такое же название должны носить посты пользователей — твуты. Для того чтобы видеть твуты других пользователей, на этих пользователей нужно *подписаться*.

Джо продумал несколько сценариев пользовательского поведения. Это функционал, который нам нужно реализовать, чтобы Джо смог достичь своей цели и помочь людям упростить общение.

- Пользователи авторизуются в Twootr при помощи уникального ID и пароля.
- У каждого пользователя есть список подписок, то есть список пользователей, на которых он подписан.
- При публикации пользователем твутов все его подписчики, авторизованные в системе, должны немедленно их видеть.
- После авторизации пользователь должен видеть твуты всех своих подписчиков с момента своего последнего посещения.

- Все пользователи должны иметь возможность удалять свои твуты. Удаленные твуты больше не видны подписчикам.
- У пользователей должна быть возможность авторизоваться как с мобильного телефона, так и через веб-сайт.

Предлагаем начать обсуждение методов реализации решений, удовлетворяющих требованиям Джо, с краткого обзора целостной картины технических решений, с которыми нам придется столкнуться.

Обзор разработки

Если в какой-то момент вам понадобится ознакомиться с исходными кодами программ данной главы, вы всегда можете найти их в репозитории: `/com/iteratrlearning/shu_book/chapter_06`.
Если вы захотите посмотреть на проект в действии, вам нужно запустить класс `Twootr Server` в IDE, а затем в браузере перейти по адресу: *http://localhost:8000*.

Если вы обратитесь к последнему требованию и подумаете над ним, оно наверняка вас очень удивит, поскольку в отличие от всех остальных систем, описанных в данной книге, на этот раз вам нужно построить систему, состоящую из множества компьютеров, общающихся между собой определенным образом. Все потому, что ваши пользователи должны иметь возможность запускать программное обеспечение на разных устройствах. Например, один пользователь загружает Twootr через веб-сайт на своем домашнем компьютере, а другой — на мобильном телефоне. Как этим пользовательским интерфейсам общаться друг с другом?

Распространенным подходом при решении подобного рода задач разработчиками является применение *клиент-серверной* модели. При таком подходе к разработке мы разделяем все компьютеры на две основных группы. У нас есть клиенты, которые запрашивают какие-то услуги, и есть *серверы*, которые эти услуги предоставляют.

В нашем случае клиентами могут быть веб-сайты или мобильные приложения, предоставляющие пользовательский интерфейс, через который можно взаимодействовать с сервером Twootr. Сервер же обрабатывает практически всю логику, отправляет и принимает твуты от различных клиентов. Это показано на рис. 6-1.

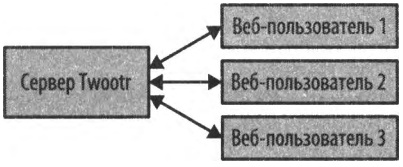

Рис. 6-1. Клиент-серверная модель

Из требований и разговора с Джо нам ясно, что важной частью данной системы является необходимость незамедлительно видеть твуты, публикуемые пользователями, на которых вы подписаны. Значит, пользовательский интерфейс должен иметь возможность принимать твуты с сервера и отправлять их на сервер. В рамках общей картины есть два различных стиля взаимодействия, которые можно использовать для достижения этой цели: push или pull.

Технология Pull

Технология Pull подразумевает, что клиент отправляет запросы на сервер и запрашивает в ответ информацию. Данный способ сетевой коммуникации часто называют методом «из точки-в точку» или технологией «запрос-ответ». Это практически самый распространенный метод, он используется в большинстве сетевых приложений. Когда вы загружаете веб-сайт, он делает HTTP-запрос на определенный сервер и получает страницы данных. Технология Pull удобна, когда клиент сам управляет тем, какая информация ему нужна. Например, если вы просматриваете Википедию, вы управляете тем, какие страницы вам интересны, и получаете ответ с интересующим вас содержимым. Схема, иллюстрирующая работу технологии Pull, представлена на рис. 6.2.

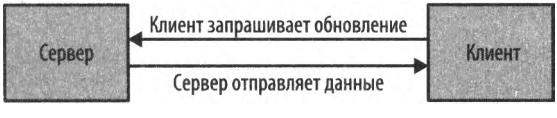

Рис. 6-2. Технология Pull

Технология Push

Другой подход — это *технология Push*. К ней стоит относиться как к реактивному или событийно-ориентированному способу коммуникации. В данной модели поток событий инициируется публикатором, а подписчики

лишь принимают его контент. Таким образом, каждое подключение (соединение) вместо схемы «один-к-одному» представляет собой схему «один-ко-многим». Эта модель очень удобна в системах, где различным компонентам необходимо поддерживать связь в рамках непрерывной схемы соединения множественных событий. Например, если речь идет о проектировании фондовой биржи, различные компании предпочтут видеть обновление цен или показателей постоянно, а не отправлять запрос при каждой необходимости. Схема, иллюстрирующая принцип технологии Push, показана на рис. 6-3.

Рис. 6-3. Технология Push

В случае с Twootr событийно-ориентированная модель связи выглядит более подходящей, так как по большому счету система состоит из потока твутов. В нашей модели события — это и есть твуты. Мы определенно могли бы разрабатывать приложение в стиле запрос-ответ. Однако если мы пойдем этим путем, клиенту придется постоянно отправлять запросы на сервер вроде: «Эй, кто-нибудь твутил с момента моего последнего запроса?» При событийно-ориентированной схеме клиент просто подписывается на события. То есть подписывается на пользователя, а сервер «проталкивает» ему нужные твуты.

Выбор событийно-ориентированной системы влияет на разработку всего остального приложения. При написании реализации главного класса приложения мы будем принимать события и отправлять их. Технология приема и отправки события определяет шаблоны, используемые в нашем коде, а также процесс написания тестов.

От событий к разработке

Говоря о том, что мы разрабатываем клиент-серверное приложение, стоит отметить, что в этой главе мы будем больше заниматься серверной частью, нежели клиентской. Как может выглядеть код клиента вы увидите в разделе «Пользовательский интерфейс» в главе 7. Есть несколько причин, почему мы будем заниматься именно серверной частью. Во-первых, эта книга о том, как разрабатывать программное обеспечение на Java, которое весьма

активно используется на стороне сервера и не так активно на стороне клиента. Во-вторых, в серверной части находится вся основная логика приложения — его мозг. В клиентской части нет ничего сложного, ее нужно просто увязать с пользовательским интерфейсом для публикации и подписки на события.

Связь

Установив, что мы хотим отправлять и принимать сообщения, давайте перейдем к следующему шагу и займемся подбором технологий, подходящих для реализации отправки сообщений клиенту или серверу от клиента. И тут перед нами встает проблема выбора. Есть несколько путей, которыми мы можем пойти:

- WebSockets — современный и легкий протокол для обеспечения дуплексной (двунаправленной) связи событий через поток TCP. Он часто используется для событийно-ориентированной связи между браузером и веб-сервером. Кроме того, он поддерживается последними версиями браузеров.
- Набирают популярность облачные очереди сообщений вроде Amazon Simple Queue Service, применяющихся для отправки, хранения и получения сообщений. Очередь сообщений — это способ представления внутренней связи путем отправки сообщений, которые могут быть получены одним процессом из группы процессов. Плюсом облачного размещения сервиса является то, что вашей компании не нужно заботиться и прилагаться усилия в этом направлении (размещение оборудования).
- Есть много хороших открытых транспортеров или очередей сообщений, таких как Aeron, ZeroMQ и AMPQ. Многие из них избегают привязки к поставщику, хотя могут ограничить ваш выбор клиента чем-то, что может взаимодействовать с очередью сообщений. Например, не будут корректно работать, если клиент — это веб-браузер.

Это далеко не полный список. Как вы можете видеть, различные технологии имеют свои недостатки и профиль использования. Вполне вероятно, одна из них может подойти для нашего проекта. Попробуйте сделать выбор. Возможно, через какое-то время, поразмыслив, вы поймете, что сделали неправильный выбор, и решите выбрать что-нибудь другое. Не исключено также, что вы захотите воспользоваться разными технологиям для разных типов клиентов. В любом случае выбирать одну из данных технологий в самом

начале проектирования, а потом принудительно оставаться с ней — не самое удачное архитектурное решение. Чуть позже в этой главе мы расскажем вам, как абстрагироваться от конкретных архитектурных решений и тем самым избежать критической ошибки при проектировании.

Возможен подход, при котором вы будете комбинировать различные технологии взаимодействия. Например, использовать различные технологии для различных типов подключенных клиентов. На рис. 6-4 показано применение WebSockets для взаимодействия с веб-сайтом, а также Android Push-уведомлений для мобильного приложения Android.

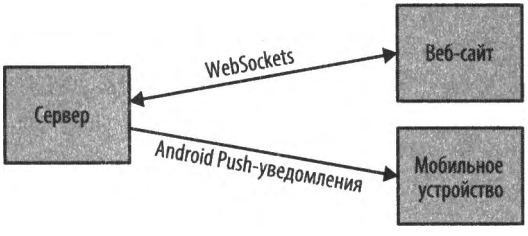

Рис. 6-4. Различные коммуникационные подходы

GUI — графический интерфейс пользователя

Объединение выбора технологии взаимодействия или пользовательского интерфейса с ядром серверной логики также имеет несколько недостатков:

- Это сложное решение, которое непросто тестировать. Каждый тест должен проводить проверку системы путем публикации и подписки на события, работая параллельно с основным сервером.
- Это нарушает принцип единственной ответственности, о котором мы говорили в главе 2.
- Это предполагает, что наш пользовательский интерфейс будет выступать в роли клиента, что, возможно, неплохо для Twootr, однако в прекрасном будущем мы, вероятно, захотим иметь чат-бот с искусственным интеллектом, который будет помогать пользователям решать проблемы. Или хотя бы отсылать GIF-ки с котиками.

Соответственно, нам необходимо проявить благоразумие в применении абстракции для развязывания отправки сообщений в интерфейсе от логического ядра приложения. Нам нужен интерфейс, через который мы могли бы отправлять сообщения клиенту, и интерфейс, через который мы могли бы получать сообщения от клиента.

Продолжаем

Другая сторона приложения рождает похожие вопросы. Как нам хранить данные из Twootr? Есть несколько вариантов:

- Обычный текстовый файл, который мы можем индексировать и осуществлять по нему поиск. С него просто считывать записанную информацию и можно избавиться от зависимости от другого приложения.
- Традиционная база данных SQL. С ней легко разобраться и ее легко тестировать. Имеется поддержка запросов.
- База данных NoSQL. Существует много различных баз данных с различным назначением, языками запросов и моделями хранения данных.

Мы точно не знаем, что именно нам выбрать перед тем, как приступить к разработке. Особенно при учете, что со временем наш проект может расширяться. Мы действительно хотим отделить систему хранения данных от остального приложения. Между этими двумя задачами есть нечто общее. Они обе связаны с тем, чтобы не привязываться к каким-то конкретным технологиям.

Гексагональная архитектура

По правде говоря, есть название для более общей архитектуры, которая могла бы помочь нам решить нашу проблему. Она называется *архитектурой портов и адаптеров* или *гексагональной архитектурой* и была впервые предложена Алистером Кокберном[1]. Идея подхода показана на рис. 6-5. Ядром приложения является его основная логика, которую вы прописываете. Вы хотите сохранить разделение между различными технологиями реализации узлов и этим ядром.

Всегда, когда вы хотите отвязать какое-то конкретное технологическое решение от ядра (логики) приложения, вы сталкиваетесь с портом. События из внешнего мира прибывают и отправляются из логического ядра через порт.

Адаптер — это ориентированный на конкретную технологию реализованный код, взаимодействующий с портом. Например, у нас есть порт для публикации подписки на события в пользовательском интерфейсе, и WebSocket-адаптер, взаимодействующий с веб-браузером.

[1] Подробнее о гексагональной архитектуре можно почитать на https://oreil.ly/wJO17. — *Прим. авт.*

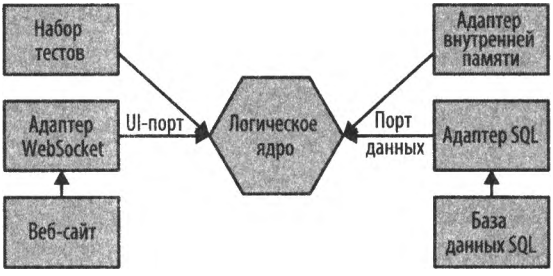

Рис. 6-5. Гексагональная архитектура

В системе есть и другие компоненты, для которых вы могли бы создать порт и адаптер. Еще один момент, о котором стоит позаботиться для расширенной версии Twootr, — это система уведомлений. Информировать пользователей о том, что у них накопилось много твутов, чтобы заинтересовать их и заставить авторизоваться, можно тоже при помощи порта. Вероятно, вы захотите реализовать его вместе с адаптером для уведомлений по электронной почте или текстовых сообщений.

Другой пример порта, приходящий на память, это сервисы аутентификации. Можно начать с адаптера, который просто хранит имена пользователей и пароли, а потом подставляет их при помощи системы OAuth или применяет их в какой-нибудь другой системе. В версии Twootr, рассматриваемой нами в данной главе, мы не будем заходить настолько далеко, чтобы делать абстракцию аутентификации. Потому что требования к системе и изначальное обдумывание проекта не выявили такой необходимости.

Возможно, пока вы не очень понимаете, что стоит отнести к порту, а что к основному домену. Вы можете впасть в крайность и «завести» сотни или даже тысячи портов в приложении, и практически все это может быть абстрагировано от домена. В другой крайности у вас может не быть ничего. Решение о том, где на этой шкале расположить ваше приложение, является сугубо личным и зависит от ваших предпочтений и локальных обстоятельств. Здесь нет правил.

Чтобы определиться с решением, подумайте обо всем, что является критически важным с точки зрения решаемой бизнес-проблемы, как о жизни внутри ядра приложения; и обо всем, что является специфической технологией или средством связи с окружающим миром, как о жизни вне этого ядра. Вот принцип, которым мы руководствовались при разработке данного приложения. Итак, бизнес-логика — это часть нашего домена, являющегося сердцем приложения, в то время как ответственность за живучесть

и событийно-ориентированную связь с пользовательским интерфейсом возложена на порты.

С чего начать

Можно было бы заниматься планированием разработки на данном этапе более детально, проектируя развернутые схемы и рассуждая о том, какую функциональность должен нести в себе каждый класс. Однако мы никогда не считали такой подход к разработке программного обеспечения чрезвычайно продуктивным. Он ведет к большому числу предположений и проектных решений, изложенных в виде маленьких «ящиков» на структурной схеме приложения. А такие схемы обычно бывают не самыми маленькими. Однако нырять в программирование, не обдумав общую концепцию приложения, тоже не лучшее решение. Чтобы избежать хаоса, разработка программного обеспечения требует некоторой начальной продуманности, однако если сосредоточиться только на проектировании без реального программирования, можно довольно быстро прийти к бесплодному и нереалистичному проекту.

Подход, подразумевающий тщательное проектирование перед непосредственно программированием, называется *BDUF* (*Big Design Up Front* — изначальное длительное проектирование). BDUF часто противопоставляется Agile или итеративной методике, приобретшей популярность за последние 10–20 лет. Так как мы считаем итеративный подход более эффективным, то описание процесса разработки в последующих разделах будет вестись именно в такой манере.

В предыдущей главе вы познакомились с TDD — разработкой через тестирование. Соответственно, вы вряд ли удивитесь, если мы предложим начать разработку проекта с создания тестового класса `TwootrTest`. Поэтому давайте начнем с тестирования авторизации пользователя: тест `shouldBeAble ToAuthenticateUser()`. В данном тесте пользователь будет авторизовываться и корректно аутентифицироваться. В примере 6-1 показан скелет метода.

Пример 6-1. Скелет метода `shouldBeAbleToAuthenticateUser()`

```
@Test
public void shouldBeAbleToAuthenticateUser()
{
    // receive logon message for valid user
```

```
    // logon method returns new endpoint.

    // assert that endpoint is valid
}
```

Вообще, чтобы реализовать этот тест, нам нужно создать класс Twootr и обзавестись возможностью моделирования события авторизации. Условимся, что в данной главе любой метод, связанный с каким-либо событием, будет иметь префикс `on`. Например, мы собираемся создать метод с названием `onLogon`. Однако какой будет сигнатура этого метода? Какую информацию он должен принимать в качестве параметров и какую информацию возвращать?

Мы уже приняли архитектурное решение, позволяющее отделить слой пользовательского интерфейса с портом. Поэтому теперь нужно принять решение о реализации API. Нам нужен способ передачи событий пользователю. К примеру, если другой пользователь, на которого подписан данный пользователь, публикует твут. Нам также нужен способ получения событий от данного пользователя. В Java мы можем просто использовать метод, предназначенный для представления событий. Таким образом, если UI-адаптер (адаптер пользовательского интерфейса) захочет опубликовать событие в Twootr, он будет вызывать метод определенного объекта, расположенного в ядре системы. Если Twootr хочет опубликовать событие, он будет вызывать метод определенного объекта, расположенного в адаптере.

Целью портов и адаптеров является отделение конкретного адаптера от ядра системы. Это значит, что нам нужен некий способ абстрагирования от различных адаптеров — интерфейс. Мы могли бы воспользоваться и абстрактным классом. Такое решение тоже работало бы, однако в нашей ситуации применение интерфейсов более рационально, потому что классы реализовали бы больше, чем один интерфейс. Кроме того, за счет применения интерфейса мы ограждаем себя в будущем от дьявольского искушения добавить какое-нибудь состояние в API. Введение состояний в API — плохая идея, поскольку различные реализации адаптеров могут представлять свое внутреннее состояние различными способами. Поэтому размещение состояния в API приведет к связанности.

Нам не нужно использовать интерфейс для объектов, в которых публикуются пользовательские события, потому что для них будет только одна реализация в ядре — тут мы можем использовать обычный класс. Визуальное представление этого подхода показано на рис. 6-6. Конечно, нам нужно имя, или даже пара имен, для представления API для отправки и получения событий.

Здесь есть много вариантов. На практике подойдет любое название, которое ясно дает понять, что эти API принимают и отправляют события.

Мы остановились на `SenderEndPoint` для класса, который отправляет события в ядро, и на `ReceiverEndPoint` для интерфейса, который принимает события из ядра. По сути, мы могли бы поменять местами обозначения отправителя и приемника событий в зависимости от точки зрения пользователя или адаптера. Здесь мы преимущественно считаем ядро первым, адаптер вторым.

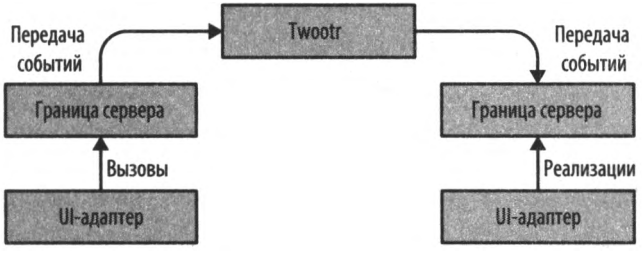

Рис. 6-6. События в коде

Определившись с направлением, можно написать тест `shouldBeAbleToAuthenticateUser()`. Он нужен для проверки того, что при попытке авторизации с корректным именем пользователя авторизация проходит. Что здесь подразумевается под авторизацией? Что ж, мы хотим возвращать объект `SenderEndPoint`, поскольку это объект, возвращаемый пользовательскому интерфейсу, представляющий пользователя, который авторизуется. Затем к классу Twootr нам нужно добавить метод, представляющий событие авторизации, что позволит тесту успешно компилироваться. Сигнатура нашей реализации показана в примере 6-2. Поскольку TDD подразумевает наличие минимальной реализации для того, чтобы тесты успешно выполнялись, а затем уже развивалась основная реализация, то мы просто создадим экземпляр объекта `SenderEndPoint` и вернем его из метода.

Пример 6-2. Первая сигнатура `onLogon`

```
SenderEndPoint onLogon(String userId, ReceiverEndPoint receiver);
```

После успешной компиляции можно написать другой тест — `shouldNotAuthenticateUnknownUser()`. Он позволит нам убедиться, что пользователь, не известный системе, не сможет авторизоваться. При написании этого теста возникает интересная ситуация. Как смоделировать провальный сценарий?

В данном случае мы не хотим возвращать объект SenderEndPoint, но при этом нам нужен способ передачи информации пользовательскому интерфейсу о том, что авторизация провалена. Один из вариантов — применить исключения, о которых мы говорили в главе 3.

Исключения решили бы задачу, но, пожалуй, это не совсем соответствовало бы концепции. Неудачная авторизация — на самом деле не совсем правильный сценарий для возникновения исключения. Это то, что происходит всегда. Люди ошибаются при вводе логина, ошибаются при вводе пароля, иногда они просто путают веб-сайты. Общепринятым можно назвать подход, при котором мы бы возвращали объект SenderEndPoint при удачной авторизации, и null при неудачной. Но это тоже не совсем корректный способ по нескольким причинам.

- Если другой разработчик использует значение без проверки, что оно не null, то он получит исключение NullPointerException. Такой род ошибок наиболее часто встречается в среде Java-разработчиков.
- Подобные ошибки невозможно выявить во время компиляции. Они возникают во время выполнения.
- Из сигнатуры метода невозможно сказать, в каких ситуациях он преднамеренно возвращает null, а в каких ситуациях это результат бага.

Лучший подход в нашем случае — использовать тип данных Optional. Такая возможность появилась в Java 8. Optional моделирует значения, которые либо присутствуют, либо отсутствуют. Это исходный (generic) тип, его можно представить в виде ящика, в котором значение или есть, или нет. Это коллекция только с одним значением или без значений вовсе. Использование Optional в качестве возвращаемого значения дает однозначное представление о том, что происходит при неудачной авторизации. Метод возвращает пустой объект Optional. В данной главе мы будем много говорить о том, как создавать и правильно использовать объект Optional. Теперь мы можем преобразовать наш метод onLogon таким образом, как показано в примере 6-3.

Пример 6-3. Вторая сигнатура onLogon:

```
Optional<SenderEndPoint> onLogon(String userId, ReceiverEndPoint receiver);
```

Нам также нужно модернизировать shouldBeAbleToAuthenticateUser() тест, чтобы он проверял, присутствует ли значение Optional. Наш следующий тест — shouldNotAuthenticateUserWithWrongPassword() показан в примере 6-4.

Он проверяет, что пользователь, который авторизуется, вводит корректный пароль. Все это означает, что наш метод `onLogon()` должен хранить не только имена пользователей, но и их пароли в `Map`.

Пример 6-4. shouldNotAuthenticateUserWithWrongPassword

```
@Test
public void shouldNotAuthenticateUserWithWrongPassword()
{
    final Optional<SenderEndPoint> endPoint = twootr.onLogon(
        TestData.USER_ID, "bad password", receiverEndPoint);

    assertFalse(endPoint.isPresent());
}
```

Для хранения данных логично было бы использовать `Map<String, String>`, где ключом был бы ID пользователя, а значением — пароль. На самом деле вопрос пользователя очень важен для нашего домена. У нас есть пользовательские истории и системный функционал, связанный с общением пользователей между собой. Настало время добавить в реализацию доменный класс User. Наша структура данных модифицируется к `Map<String, User>`, где ключ — ID пользователя, а значение — рассматриваемый объект `User`.

Существует достаточно популярное мнение, что TDD в некоторой степени препятствует разработке непосредственно программного обеспечения. То есть вынуждает вас писать тесты, а в итоге вы получаете слабую доменную модель и необходимость переписывать саму реализацию. Под *слабой доменной моделью* мы подразумеваем модель, в которой доменные объекты не имеют достаточно основных алгоритмов и в основном состоят из различных методов, выполненных в процедурном стиле. Это утверждение можно считать достаточно справедливым по отношению к способу, которым зачастую применяют TDD. Подобрать правильный момент для добавления доменного класса или реализации какого-нибудь реального элемента программы — довольно деликатная задача. Если данный элемент программы — это то, чем часто пользуются, то вам действительно стоит подумать над тем, чтобы внести его в доменную часть.

Однако и здесь есть несколько антишаблонов, о которых стоит упомянуть. К примеру, если вы создали несколько структур поиска, использующих одинаковый ключ в один и тот же момент, но связанный с разными значениями, то вам не хватает доменного класса. То есть если мы отслеживаем некий

набор подписчиков и пароли пользователей и у нас есть два объекта `Map`, завязанных на пользовательские ID (один для подписчиков, а другой для пароля), то нам не достает какой-то конструкции в домене. Наш класс `User` мы представили только с одним значением, которое нас интересует, — паролем. Понимание доменной проблемы говорит нам о том, что пользователь — это важный элемент системы, поэтому в данном случае мы не были слишком преждевременны.

Пароли и безопасность

До этого момента мы не говорили о безопасности вообще. На самом деле не говорить о проблемах безопасности и надеяться, что они исчезнут сами по себе — это любимая стратегия в индустрии технологий. Научить вас писать безопасный код — это не первичная, и даже не вторичная цель данной книги. Однако в Twootr мы используем и храним пароли для аутентификации пользователей, поэтому нам все-таки стоит немного обдумать этот вопрос.

Самый простой способ хранения паролей — представить их, как и все остальные данные, в виде `String`, предназначенных для хранения *простого текста*. Это в корне плохой подход. Потому что он предполагает, что любой, у кого есть доступ к вашей базе данных, имеет доступ к паролям всех ваших пользователей. Злоумышленники могут (и зачастую так и делают) использовать простые текстовые пароли для входа в систему. Кроме того, многие люди задают одинаковые пароли для различных сервисов. Если вы не верите нам — спросите у своих родственников постарше!

Чтобы исключить доступ к паролям для всех, у кого есть доступ к базе данных, вы можете воспользоваться *криптографической хэш-функцией*. Она принимает на вход строку заданного размера и конвертирует ее в выходное значение, называемое дайджест (digest). Криптографические хэш-функции детерминированы, то есть если вы пропустите через функцию одно и то же значение дважды, то получите одинаковый результат. Это необходимо для того, чтобы иметь возможность проверить хэшированный пароль в будущем. Другая особенность функции состоит в том, что если прямое преобразование (из пароля в дайджест) выполняется достаточно быстро, то обратное (дайджест в пароль) занимает очень много времени или очень много ресурсов, что практически невозможно для злоумышленника.

Разработка криптографических хэш-функций сейчас активно развивается. Правительства и компании вкладывают в это много денег. Реализовать

хэш-функцию правильно очень сложно, поэтому вы вряд ли создадите свою собственную. В Twootr мы используем библиотеку под названием Bouncy Castle[1] с открытым исходным кодом. Она подвергалась серьезной экспертной оценке. Предлагаем воспользоваться в Twootr хэш-функцией *Scrypt*, которая представляет собой современный алгоритм, разработанный специально для работы с паролями. В примере 6-5 показан пример кода.

Пример 6-5. KeyGenerator

```
class KeyGenerator {
    private static final int SCRYPT_COST = 16384;
    private static final int SCRYPT_BLOCK_SIZE = 8;
    private static final int SCRYPT_PARALLELISM = 1;
    private static final int KEY_LENGTH = 20;

    private static final int SALT_LENGTH = 16;

    private static final SecureRandom secureRandom = new SecureRandom();

    static byte[] hash(final String password, final byte[] salt) {
        final byte[] passwordBytes = password.getBytes(UTF_16);
        return SCrypt.generate(
            passwordBytes,
            salt,
            SCRYPT_COST,
            SCRYPT_BLOCK_SIZE,
            SCRYPT_PARALLELISM,
            KEY_LENGTH);
    }

    static byte[] newSalt() {
        final byte[] salt = new byte[SALT_LENGTH];
        secureRandom.nextBytes(salt);
        return salt;
    }
}
```

Основная проблема большинства хэш-функций в том, что, хотя они и требуют много ресурсов для обратного вычисления, обратную функцию можно получить путем принудительного перебора паролей до определенной

[1] https://www.bouncycastle.orgwww.bouncycastle.org. — *Прим. авт.*

длины или при помощи радужной таблицы. Чтобы обезапасить себя от такого варианта развития событий, мы используем соль. *Соли* — это дополнительные случайно сгенерированные входные данные, которые добавляются к криптографической хэш-функции. За счет добавления к каждому паролю дополнительных входных данных, генерирующихся автоматически, мы исключаем возможность поиска обратной функции. Ведь для этого злоумышленникам нужно будет знать и хэш-функцию, и соль.

Итак, обсуждая вопрос хранения паролей, мы упомянули несколько базовых принципов безопасности. На самом деле поддержка безопасности системы — это непрерывная работа. Однако нужно думать не только о безопасности хранимых данных, но также и о безопасности передаваемых данных. Когда кто-либо подключается к вашему серверу при помощи клиента, ему нужно передать пользовательский пароль через сетевое соединение. Если злоумышленник перехватит это соединение, то у него будет доступ к копии пароля, и тогда он сможет совершить что-нибудь подлое, захватив чужую учетную запись и общаясь от чужого имени!

В случае с Twootr сообщение об авторизации мы получаем через WebSockets. Значит, чтобы обеспечить безопасность WebSocket соединения, нужно защитить его от атаки посредника (так называемой атаки «человек посередине»). Есть несколько способов это сделать. Самый простой — использовать *TLS* (*Transport Level Security* — безопасность уровня передачи). Это криптографический протокол, направленный на обеспечение приватности и интеграции данных по отношению к данным, передаваемым через него.

Компании с серьезным подходом к безопасности организуют регулярные проверки и проводят анализ своего программного обеспечения. Можно периодически привлекать внештатных консультантов или задействовать своих же специалистов, дав им задание проникнуть в систему безопасности под видом злоумышленника.

Подписчики и твуты

Следующее требование, которое нам нужно реализовать — подписчики. В данном случае процесс разработки программного обеспечения можно представить одним из двух способов. Один из них называется «снизу вверх» (bottom-up): он начинается с разработки ядра приложения (модели хранения данных или взаимосвязей между доменными объектами). После чего происходит переход к построению функционала системы. При таком подходе

система подписок сводится к вопросу о том, как моделировать взаимоотношения между пользователями. Это определенно взаимоотношения типа «множество к множеству», поскольку у пользователя может быть много подписчиков, да и сам он может быть подписан на многих других пользователей. Затем мы доходим до верхнего уровня данной модели — уровня функционала, предназначенного для того, чтобы сделать пользователей довольными.

Другой подход разработки — *«нисходящий»*. Он начинается с требований пользователей и попытки разработать поведение или функционал, необходимый для реализации этих требований, постепенно продвигаясь вглубь к вопросам хранения и модели данных. Например, мы можем начать с разработки API для получения события подписки на другого пользователя, а затем заняться механизмом хранения, необходимого для этой задачи, постепенно продвигаясь от API к логике приложения.

Нельзя сказать, что один подход лучше при любых условиях, а другого следует избегать. Однако, что касается приложений линейного типа, для написания которых очень часто используется язык Java, наш опыт свидетельствует о том, что лучше применять подход «сверху вниз». Дело в том, что при разработке модели данных или доменного ядра у вас появляется большой соблазн потратить много времени на создание совершенно ненужных возможностей. Негативная же сторона подхода «сверху вниз» в том, что, когда вы выполняете много требований, ваш изначальный дизайн может оказаться неудовлетворительным. Это значит, что вам нужно сохранять бдительность и придерживаться итеративного подхода при разработке, подразумевающего постепенное наращивание вашего приложения.

В этой главе мы продемонстрируем вам нисходящий подход и начнем с написания теста для проверки функциональности механизма подписок (пример 6-6). Итак, наш пользовательский интерфейс должен отправлять нам событие, показывающее, что пользователь хочет подписаться на другого пользователя. Так что наш тест будет вызывать метод `onFollow` с уникальным ID пользователя, на которого организуется подписка, в качестве аргумента. Конечно, данный метод пока еще не существует. Поэтому нам нужно объявить его в классе Twootr, чтобы код компилировался.

Моделирование ошибок

Тест в примере 6-6 охватывает основную часть операции подписки, поэтому нам нужно убедиться, что операция проходит нормально.

Пример 6-6. shouldFollowValidUser

```
@Test
public void shouldFollowValidUser()
{
    logon();

    final FollowStatus followStatus = endPoint.onFollow(TestData.OTHER_USER_ID);

    assertEquals(SUCCESS, followStatus);
}
```

Теперь у нас есть удачный сценарий, однако нам стоит учесть еще несколько вариантов развития событий. Что если переданный в качестве аргумента ID пользователя не связан с реальным пользователем. Или пользователь, запрашивающий подписку, уже подписан на данного пользователя. Нам нужно смоделировать различные результаты или состояния, которые может возвращать наш метод. Существует несколько вариантов, которыми мы можем воспользоваться.

Один из способов — генерировать исключение, когда операция завершается и возвращать `void`, если она прошла успешно. Этот способ вполне нас устроит. Он не противоречит идее о том, что исключения должны использоваться только для управляющего потока исключений, в том смысле, что правильно спроектированный пользовательский интерфейс должен избегать появления таких сценариев при нормальных обстоятельствах. Давайте рассмотрим некоторые альтернативы, которые работают со статусом как со значением вместо использования исключений.

Один из простых вариантов — воспользоваться булевой переменной. Значение `true` для представления успешного сценария, `false` — наоборот. Это хороший вариант в случае, если у операции есть два возможных исхода (удачно или неудачно), что позволяет быстро понять причину сбоя. Проблема переменной `boolean` проявляется в ситуациях, когда происходит сразу несколько неудачных сценариев и становится сложно разобраться в причинах.

Еще один вариант — воспользоваться простой константой `int` для представления различных неудачных сценариев. Однако, как мы уже обсуждали в главе 3 во время знакомства с исключениями, данный подход ведет к созданию ненадежного, плохо читаемого и сложного в обслуживании кода. Есть еще одна альтернатива, которая является более типобезопасной и предлагает лучшую документированность: типы `enum`. `Enum` — это список изначально

заданных констант, образующих допустимый тип. Поэтому везде, где можно использовать `interface` или `class`, можно использовать `enum`.

Перечисления (`enum`) лучше, чем основанные на `int` статусы по нескольким причинам. Если метод возвращает вам `int`, то вы не обязательно знаете, какие значения там могут содержаться. Можно добавить `javadoc` и описать возможные значения, также можно воспользоваться константами (поля `static final`), однако это все как «помада на свинье». Перечисления же могут содержать только те значения, которые указаны в объявлении `enum`. В принципе перечисления в Java могут содержать поля экземпляров и методы, объявленные для расширения функционала, хотя в нашем случае мы не будем использовать эту возможность. В примере 6-7 показано объявление статуса подписчика.

Пример 6-7. FollowStatus

```
public enum FollowStatus {
    SUCCESS,
    INVALID_USER,
    ALREADY_FOLLOWING
}
```

Так как принцип TDD обязывает нас писать простейшую реализацию для того, чтобы тест работал, соответственно, метод `onFollow` здесь должен просто возвращать значение `SUCCESS`.

Есть еще пара сценариев, о которых нужно подумать для операции `following()`. В примере 6-8 показан тест, который вынуждает нас заняться вопросом дублирования пользователей. Для этого нам нужно добавить в класс `User` набор пользовательских ID, которые будут представлять набор пользователей, на которых подписан наш пользователь, и позволят убедиться, что процесс добавления не дублируется. Сделать это очень просто при помощи Java коллекций. У нас уже есть интерфейс `Set`, который объявляет уникальные элементы, а метод add будет возвращать `false`, если добавляемый элемент уже существует в `Set`.

Пример 6-8. shouldNotDuplicateFollowValidUser

```
    @Test
    public void shouldNotDuplicateFollowValidUser()
    {
        logon();
```

```
        endPoint.onFollow(TestData.OTHER_USER_ID);

        final FollowStatus followStatus = endPoint.onFollow(TestData.OTHER_USER_ID);
        assertEquals(ALREADY_FOLLOWING, followStatus);
    }
```

Тест `shouldNotFollowInValidUser()` делает утверждение о том, что если пользователь не действителен, то возвращаемый статус это покажет. Он похож на `shouldNotDuplicateFollowValidUser()`.

Твутинг

Разобравшись с основами, давайте займемся самой интересной частью продукта — твутингом! Из требований к продукту мы знаем, что любой пользователь может опубликовать твут и все авторизованные на данный момент подписчики должны немедленно его увидеть. На практике мы не можем точно знать, что пользователи увидели твут немедленно. Возможно, они зашли со своего компьютера, но отошли выпить кофе, или открыли другую вкладку или, не дай бог, работают.

Теперь вы уже знакомы с общим подходом. Нам нужно написать тест для сценария, когда авторизованный пользователь получает твут от другого пользователя. Назовем его `shouldReceiveTwootsFromFollowedUser()`. Кроме авторизации и подписки, для этого теста требуются еще кое-какие элементы.

Во-первых, нам нужно смоделировать отправку твута, добавив метод `onSendTwoot()` к `SenderEndPoint`. У него есть параметры для идентификатора твута, поэтому мы можем обратиться к нему позже.

Во-вторых, нам нужен способ уведомления подписчиков о том, что пользователь опубликовал твут — что-то, что мы можем проверить при помощи теста. Ранее мы представляли вам `ReceiverEndPoint` как способ публикации сообщений. Настало время им воспользоваться. Давайте добавим метод `onTwoot`, как показано в примере 6-9.

Пример 6-9. ReceiverEndPoint

```
public interface ReceiverEndPoint {
    void onTwoot(Twoot twoot);
}
```

Независимо от того, какое сообщение наш UI-адаптер будет отправлять в пользовательский интерфейс, он должен проинформировать о том, что был опубликован твут. Вопрос в том, как написать тест, проверяющий, что метод `onTwoot` был вызван?

Создание моков

Самое время познакомиться с концепцией *мок*-объектов (*имитаций*). Мок-объект — это тип объекта, который претендует на роль другого объекта. У него есть все те же методы и публичные API, как у основного объекта, а для системы типов Java он выглядит как другой объект, но при этом такой же. Его задача — фиксировать любые итерации, например, вызовы методов, и иметь возможность *проверки* того, что определенный метод вызывался. Например, в данном случае мы хотим убедиться, что метод `onTwoot()` из `ReceiverEndPoint` вызывался.

Возможно, для людей с ИТ-образованием будет странным видеть слово «верифицировать» в непривычном для себя контексте. В математике и других формальных сообществах данный термин принято использовать, когда свойства системы были доказаны для всех входных значений. В мокинге это слово обозначает совсем другое. Оно подразумевает, что метод был вызван с определенным набором аргументов. Мы согласны, что можно легко запутаться, когда различные группы людей используют одно и то же слово, вкладывая в него разные значения. Поэтому советуем просто обращать внимание на контекст.

Мок-объекты могут создаваться несколькими способами. Первые мок-объекты обычно создаются вручную. По сути, мы могли бы написать реализацию мока `ReceiverEndPoint`, как показано в примере 6-10. Когда метод `onTwoot` вызывается, мы делаем соответствующую запись путем сохранения параметра `Twoot` в `List`. В результате мы можем верифицировать факт вызова функции с определенным набором аргументов путем утверждения, что в `List` содержится объект `Twoot`.

Пример 6-10. MockReceiverEndPoint

```
public class MockReceiverEndPoint implements ReceiverEndPoint
{
    private final List<Twoot> receivedTwoots = new ArrayList<>();
```

```
    @Override
    public void onTwoot(final Twoot twoot)
    {
        receivedTwoots.add(twoot);
    }

    public void verifyOnTwoot(final Twoot twoot)
    {
        assertThat(
        receivedTwoots,
        contains(twoot));
    }
}
```

На практике написание моков вручную может оказаться утомительным, кроме того, существует большая вероятность возникновения ошибок. А что нормальные программисты делают с тяжелыми и склонными к ошибкам методами? Правильно! Автоматизируют их. Есть несколько библиотек, которые могут помочь нам в создании мок-объектов. Библиотека, которой мы будем пользоваться в этом проекте, называется *Mockito*. Она находится в свободном доступе, имеет открытый исходный код и является весьма популярной. Большинство операций, связанных с Mockito, могут быть вызваны при помощи статических методов в классе `Mockito`, который мы используем в виде статического импорта. Вообще, чтобы создать мок-объект, нужно использовать метод `mock` (пример 6-11).

Пример 6-11. mockReceiverEndPoint

```
    private final ReceiverEndPoint receiverEndPoint = mock(ReceiverEndPoint.class);
```

Верификация при помощи моков

Мок-объект, который мы создали, может использоваться везде, где используется реализация `ReceiverEndPoint`. Мы можем передавать его в качестве параметра методу `onLogon()`, к примеру, для подключения UI-адаптера (адаптера пользовательского интерфейса). Если тестируемое поведение (условная часть теста — часть *when*) выполнилось, то нашему тесту нужно верифицировать, что метод `onTwoot` вызывался (часть *then*). Для того чтобы это сделать, мы «захватываем» мок-объект при помощи метода `Mockito.verify()`. Это исходный (`generic`) метод, который возвращает объект того

же типа, что и получил на входе. Мы просто вызываем рассматриваемый метод с ожидаемыми аргументами, чтобы описать ожидаемое взаимодействие с мок-объектом, как показано в примере 6-12.

Пример 6-12. verifyReceiverEndPoint

```
verify(receiverEndPoint).onTwoot(aTwootObject);
```

В последнем разделе вы могли заметить, что мы представили вам класс `Twoot`, который мы использовали в сигнатуре метода `onTwoot`. Это объект-значение, который мы будем использовать для «оборачивания» значений и представления экземпляра `Twoot`. Поскольку он будет передаваться в UI-адаптер, то должен состоять из полей с простыми значениями, а не выдавать слишком много информации из ядра домена. Например, для представления отправителя твута у нас есть `id` отправителя вместо ссылки на объект `User`. В Twoot также есть `content String` и `id` собственного объекта `Twoot`.

В данной системе объекты `Twoot` неизменяемые. Как упоминалось ранее, такой стиль снижает количество багов. Это особенно важно при работе с чем-то вроде объекта-значения, который передается в UI-адаптер. Мы хотим, чтобы наш UI-адаптер отображал `Twoot` не для того, чтобы изменять состояние `Twoot`'ов другого пользователя. Кроме того, нам ничего не стоит продолжать использовать доменный язык для именования класса `Twoot`.

Библиотеки для мокинга

В данном случае мы используем Mockito, поскольку она обладает хорошим синтаксисом и соответствует нашему любимому способу создания моков. Однако это не единственная платформа для мокинга на Java. Powermock и EasyMock тоже достаточно популярны.

Powermock может эмулировать синтаксис Mockito, однако она позволяет делать то, что не поддерживается Mockito. Например, классы `final` или статические методы. Ведутся споры о том, стоит ли тестировать с помощью мокинга такие объекты, как классы `final`. Ведь если вы не имеете возможности создать другую реализацию класса, то будете ли вы это делать в тестах? В целом использование Powermock не сильно обоснованно, однако могут возникнуть неожиданные ситуации, когда она будет полезна.

У EasyMock другой подход к мокингу. Это стилистический выбор и некоторые разработчики могут предпочесть его другим. Большая концептуальная разница в том, что EasyMock поддерживает строгий мокинг. Идея строгого мокинга состоит в том, что если вы явно не указываете, что вызов должен произойти, то это будет ошибкой. Это отражается на тестах, которые более ориентированы на поведение класса, но иногда может быть связано с лишними взаимодействиями.

SenderEndPoint

Методы `onFollow` и `onSendTwoot` объявлены в классе `SenderEndPoint`. Каждый экземпляр `SenderEndPoint` представляет собой конечную точку события отправки сообщения пользователем. За счет нашего дизайна класса `Twoot`, класс `SenderEndPoint` остается простым: он «захватывает» главный класс `Twootr` и делегирует методы, передаваемые в объекте `User` для пользователя, которого он представляет в системе. Пример 6-13 показывает объявление класса и образец одного метода, связанного с одним событием — `onFollow`.

Пример 6-13. SenderEndPoint

```
public class SenderEndPoint {
    private final User user;
    private final Twootr twootr;

    SenderEndPoint(final User user, final Twootr twootr) {
        Objects.requireNonNull(user, "user");
        Objects.requireNonNull(twootr, "twootr");

        this.user = user;
        this.twootr = twootr;
    }

    public FollowStatus onFollow(final String userIdToFollow) {
        Objects.requireNonNull(userIdToFollow, "userIdToFollow");

        return twootr.onFollow(user, userIdToFollow);
    }
```

В примере 6-13 вы могли заметить класс `java.util.Objects`. Это служебный класс, который поставляется с JDK и предлагает удобные методы для проверки нулевых ссылок и реализации методов `hashCode()` и `equals()`.

Существуют альтернативы использованию SenderEndPoint, о которых стоит поговорить. Мы можем получать события, связанные с пользователем, раскрывая методы объекта Twootr и ожидая, что любой UI-адаптер вызывает эти методы напрямую. Это довольно субъективный вопрос, в общем, как и многое в программировании. Некоторые люди могут расценивать создание SenderEndPoint как ненужное усложнение программы.

Главная мотивация здесь, как уже упоминалось ранее, в том, что мы не хотим раскрывать доменный объект User в UI-адаптере, а хотим только взаимодействовать с ним при помощи простых событий. Было бы возможно использовать идентификатор пользователя в качестве параметра для всех методов-событий Twootr. Но тогда первым шагом для любого события нужно было бы обращаться к объекту User через идентификатор, несмотря на то, что он уже содержится в SenderEndPoint. Такое решение привело бы к ненужности SenderEndPoint, но добавило бы много другой работы и трудностей.

В целом, чтобы отправить Twoot, нам нужно немного изменить код доменного ядра. У объектов User должен быть набор подписчиков, прикрепленных к ним, которые должны получать уведомления о новых твутах. В примере 6-14 представлен наш код для метода onSendTwoot. Он определяет авторизованных пользователей и выдает им уведомление о приеме твута. Если вы не знакомы с методами filter и forEach или с синтаксисом :: и ->, не волнуйтесь, мы вернемся к ним в разделе «Функциональное программирование».

Пример 6-14. onSendTwoot

```
void onSendTwoot(final String id, final User user, final String content)
{
    final String userId = user.getId();
    final Twoot twoot = new Twoot(id, userId, content);
    user.followers()
        .filter(User::isLoggedOn)
        .forEach(follower -> follower.receiveTwoot(twoot));
}
```

В объекте User также необходимо реализовать метод receiveTwoot(). Как User получает твут? При помощи пользовательского интерфейса он должен уведомить пользователя о том, что появился твут, готовый к демонстрации, за счет генерации события, которое провоцирует вызов receiverEndPoint.onTwoot(twoot). Это вызов метода, который мы верифицировали при помощи мокинга. За счет вызова метода тест проходит успешно.

Последний вариант теста представлен в примере 6-15 и именно этот код вы увидите, если скачаете проект с GitHub. Вы можете заметить, что код несколько отличается от того, что мы описывали ранее. Во-первых, поскольку тесты приема твутов уже написаны, некоторые операции были преобразованы в общие методы. Например, метод `logon()`, который авторизует нашего первого пользователя в системе, который является входной частью для многих тестов. Во-вторых, тест также создает объект `Position` и передает его в `Twoot`. Кроме того, он верифицирует взаимодействие с хранилищем `twootRepository`. Что еще за хранилище? Обе эти конструкции не были нам нужны до этого момента, но являются важной частью развития нашей системы. Поэтому подробнее о них мы расскажем в двух следующих разделах.

Пример 6-15. shouldReceiveTwootsFromFollowedUser

```
    @Test
    public void shouldReceiveTwootsFromFollowedUser()
    {
        final String id = "1";

        logon();

        endPoint.onFollow(TestData.OTHER_USER_ID);

        final SenderEndPoint otherEndPoint = otherLogon();
        otherEndPoint.onSendTwoot(id, TWOOT);

        verify(twootRepository).add(id, TestData.OTHER_USER_ID, TWOOT);
        verify(receiverEndPoint).onTwoot(new Twoot(id, TestData.OTHER_USER_ID,
TWOOT, new Position(0)));
}
```

Позиции

Скоро вы познакомитесь с объектами `Position`. Но перед тем как рассказать вам об их содержании, мы хотим обсудить с вами их предназначение. Итак, нам необходимо воплотить следующее требование: после авторизации пользователю должны быть предоставлены все твуты, которые он пропустил с момента предыдущей авторизации. Это подводит нас к тому, что нам нужен некий механизм повторного «воспроизведения» различных

твутов. Кроме того, нам нужно знать, какие твуты пользователь еще не видел. В примере 6-16 приведен тест для проверки данного функционала.

Пример 6-16. shouldReceiveReplayOfTwootsAfterLogoff

```
@Test
public void shouldReceiveReplayOfTwootsAfterLogoff()
{
    final String id = "1";

    userFollowsOtherUser();

    final SenderEndPoint otherEndPoint = otherLogon();
    otherEndPoint.onSendTwoot(id, TWOOT);

    logon();

    verify(receiverEndPoint).onTwoot(twootAt(id, POSITION_1));
}
```

Для того чтобы реализовать этот функционал, система должна знать, какие твуты были опубликованы за то время, что пользователя не было в системе. Есть много способов достичь данного результата. Различные подходы предлагают разные компромиссы в плане сложности реализации, правильности и производительности/масштабируемости. Поскольку мы только начинаем разрабатывать Twootr и не ожидаем большого числа пользователей, то не будем фокусироваться на масштабируемости.

- Мы могли бы отслеживать время публикации каждого твута и время оффлайна пользователя, а затем искать твуты, опубликованные за это время.
- Мы можем представить твуты в виде непрерывного потока, в котором у каждого твута есть своя позиция, и записывать позиции в момент выхода пользователя из системы.
- Мы опять же можем использовать позиции и фиксировать позицию последнего просмотренного твута.

Выбирая подходящую концепцию, стоит отойти от идеи учета сообщений по времени. Давайте представим, что в качестве единицы измерения времени мы выбрали миллисекунды. Что произойдет, если мы получим два твута в один и тот же промежуток времени? Мы не будем знать, в каком порядке

они шли. Что если твут был принят в ту же секунду, когда пользователь вышел из системы?

Фиксирование времени выхода пользователя из системы — еще одно проблемное событие. Хорошо, если пользователь все время выходит из системы путем нажатия одной кнопки. На практике это только один из способов прекратить работу с пользовательским интерфейсом. Пользователь может просто закрыть веб-браузер без непосредственного выхода из системы. Или веб-браузер может аварийно завершить свою работу. А что если пользователь был авторизован сразу из двух браузеров, а затем вышел только из одного из них? Что произойдет, если в мобильном телефоне села батарейка или пользователь закрыл приложение?

Мы выбираем самый безопасный метод определения момента повторного воспроизведения твутов: присваивать позиции твутам, а затем сохранять позицию последнего просмотренного твута. Для определения позиций вводим маленький объект-значение под названием Positioin, как показано в примере 6-17. В данном классе также есть константа для начальной позиции потока. Поскольку все наши позиции будут положительными значениями, то для начального положения мы можем использовать любое отрицательное значение. Здесь мы выбираем –1.

Пример 6-17. Position

```
public class Position {
    /**
     * Position before any tweets have been seen
     */
    public static final Position INITIAL_POSITION = new Position(-1);

    private final int value;

    public Position(final int value) {
        this.value = value;
    }

    public int getValue() {
        return value;
    }

    @Override
    public String toString() {
        return "Position{" +
```

```
            "value=" + value +
            '}';
    }

    @Override
    public boolean equals(final Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;

    final Position position = (Position) o;

    return value == position.value;
    }

    @Override
    public int hashCode() {
        return value;
    }

    public Position next() {
        return new Position(value + 1);
    }
}
```

Согласитесь, класс выглядит довольно сложным. Не так ли? Сейчас вы можете спросить себя: «Зачем мне нужны методы `equals()` и `hashCode()` вместо того, чтобы позволить Java обработать это все самостоятельно?» Что такое *объект-значение*? Почему я задаю столько вопросов? Не переживайте. Мы просто познакомили вас с новой темой и сейчас ответим на все вопросы. Зачастую очень удобно ввести маленькие объекты, представляющие значения, состоящие из нескольких полей, или дать подходящее доменное имя некоторым числовым значениям. Наш класс `Position` пример этому. Другой пример — класс `Point` (пример 6-18).

Пример 6-18. `Point`

```
class Point {
    private final int x;
    private final int y;

    Point(final int x, final int y) {
        this.x = x;
```

```
        this.y = y;
    }

    int getX() {
        return x;
    }

    int getY() {
        return y;
    }
```

Методы equals и hashcode

Если мы захотим сравнить два объекта, объявленных подобным образом, с одинаковыми значениями, то обнаружим, что на самом деле они не эквивалентны. Такая ситуация показана в примере 6-19. По умолчанию методы `equals()` и `hashCode()`, которые наследуются из `java.lang.Object`, предназначены для применения принципа эквивалентного отношения. Это значит, что если у вас есть два разных объекта, расположенных в разных местах памяти вашего компьютера, то они не эквивалентны. Даже если все их поля эквивалентны. Это может привести к множеству скрытых ошибок в программе.

Пример 6-19. Объекты `Point` не эквивалентны, хотя и должны быть

```
final Point p1 = new Point(1, 2);
final Point p2 = new Point(1, 2);
System.out.println(p1 == p2); // prints false
```

Удобно представлять себе объекты двух типов — ссылочные объекты и *объекты-значения* — в зависимости от их понятия эквивалентности. В Java у нас есть возможность переопределить метод `equals()`, чтобы задать свою собственную реализацию, которая использует поля для определения равенства значений. В примере 6-20 показана реализация для нашего класса `Point`. Мы проверяем сначала, что данные объекты относятся к одному типу, а затем что все их поля эквивалентны.

Пример 6-20. Определение эквивалентности `Point`

```
    @Override
    public boolean equals(final Object o) {
        if (this == o) return true;
        if (o == null || getClass() != o.getClass()) return false;
```

```
        final Point point = (Point) o;

        if (x != point.x) return false;
        return y == point.y;
    }

    @Override
    public int hashCode() {
        int result = x;
        result = 31 * result + y;
        return result;
    }

    final Point p1 = new Point(1, 2);
    final Point p2 = new Point(1, 2);
    System.out.println(p1.equals(p2)); // prints true
```

Взаимосвязь между equals и hashCode

В примере 6-20 мы не только переопределили метод `equals()`, но и метод `hashCode()`. Причина — во *взаимосвязи* между `equals` и `hashcode` в Java. Это значит, что если у нас есть два объекта, эквивалентных по методу `equals()`, то у `hashCode()` должен быть такой же результат. Некоторые основные Java API используют метод `hashCode()`, в особенности реализации коллекций вроде `HashMap` и `HashSet`. Они основаны на этом взаимодействии, и вы обнаружите, что они ведут себя не так, как вы этого ожидаете. Как же нам правильно реализовать `hashCode()`?

Правильные реализации не только соответствуют данному взаимодействию (контракту), но также вырабатывают значения хэш-кодов, которые равномерно распределены в диапазоне целочисленных значений (`int`). Это позволяет повысить эффективность реализаций `HashMap` и `HashSet`. Ниже приведены простые правила, которым нужно следовать, и вы получите хорошую реализацию `hashCode()`.

1. Создайте переменную `result` и присвойте ей первичное значение.
2. Возьмите каждое поле, используемое методом `equals()` и вычислите целое значение, представляющее хэш-код этого поля.
3. Объедините хэш-код поля с существующим результатом умножения предыдущего результата на первичное значение. Например, `result = 41 * result + hashCodeOfField;`.

Чтобы вычислить хэш-код каждого поля, нужно сначала определить его тип.

- Если поле представляет собой примитивное значение, то воспользуйтесь методом `hashCode()`, предоставленным в его вспомогательном классе. Например, если это тип double, то используйте `double.hashCode()`.
- Если это ненулевой объект, просто вызовите его метод `hashCode()` или используйте 0. Это может быть сокращено в методе `java.lang.Objects.hashCode()`.
- Если это массив, то вам нужно объединить значения `hashCode()` каждого из его элементов, используя те же правила, которые мы здесь описали. Для этого можно воспользоваться методами `java.util.Arrays.hashCode()`.

В большинстве случаев вам не нужно определять методы `equals()` и `hashCode()` самостоятельно. Современные Java IDE сгенерируют их для вас. Хотя заняться этим самостоятельно полезно для понимания принципов генерируемого кода. Особенно важно иметь возможность изучить пару методов `equals()` и `hashCode()` в коде, чтобы знать, насколько хорошо они реализованы.

В этом разделе мы немного поговорили об объектах-значениях, однако в будущей версии Java планируется ввести встроенные классы. Прототипы были представлены в проекте Valhalla[1]. Идея встроенных классов — обеспечить эффективный способ реализации структуры данных, которые выглядят как обычные значения. У вас все еще будет возможность использовать обычный класс, но встроенные классы будут генерировать корректные методы `hashCode()` и `equals()`, использовать меньше памяти, и в большинстве случаев будут быстрее.

При реализации этой возможности нам нужно ассоциировать `Position` с каждым объектом `Twoot`, поэтому мы добавили поле в класс `Twoot`. Нам также нужно записывать `Position` для последнего момента, когда пользователь был онлайн, поэтому в `User` мы добавили `lastSeenPosition`. Когда User принимает `Twoot`, они обновляют свой показатель `Position`. Когда пользователь авторизуется, ему передаются твуты, которые он пропустил. Так что никаких новых событий не нужно добавлять ни в `SenderEndPoint`, ни в `ReceiverEndPoint`. Повторное воспроизведение твутов также подразумевает, что объекты `Twoot` нужно где-то хранить. Изначально мы используем для этого JDK `List`. Теперь нашим пользователям не нужно быть всегда в сети, чтобы наслаждаться Twootr, что не может не радовать.

[1] https://oreil.ly/muvlT. — *Прим. авт.*

Выводы

- Вы изучили архитектурные принципы «целостной картины», а также способы взаимодействия.
- Вы узнали о возможности отделения доменной логики от библиотек и платформ.
- Вы вели разработку через тестирование, двигаясь от оболочки к ядру приложения.
- Вы применили навыки объектно-ориентированного доменного моделирования в большом проекте.

Самостоятельная работа

Если вы хотите расширить или укрепить знания, полученные в этой главе, то можете попытаться выполнить что-нибудь из этого:

- Попробуйте реализовать перенос по словам[1].
- Не читая следующую главу, составьте список того, что нужно сделать, чтобы полностью закончить Twootr.

В завершение

У нас прошла встреча с Джо, на которой мы обсудили прогресс в разработке проекта. Большинство основных требований были реализованы, и мы описали, как должна работать система в целом. Конечно, Twootr пока не закончен. Вы еще не узнали, как объединить все компоненты программы вместе, чтобы они могли обмениваться информацией, а также не познакомились с нашим подходом к хранению твутов в системной памяти так, чтобы они не пропадали при перезагрузке Twootr.

Джо действительно восхищен нашим прогрессом и очень ждет окончания работы над проектом. В последней главе мы закончим разработку Twootr и рассмотрим оставшиеся вопросы.

---

[1] https://oreil.ly/vH2Q5. — *Прим. авт.*

Расширение Twootr

Задача

Итак, ранее мы получили заказ от Джо, который хотел создать современную онлайн-платформу для общения людей. В предыдущей главе была представлена примерная концепция Twootr и реализация основной части доменной логики приложения. Разработка велась через тестирование. Вы познакомились с некоторыми решениями в области разработки и моделирования данных, научились решать и структурировать задачи. Однако пока этого недостаточно для завершения проекта Twootr, поэтому в этой главе мы продолжим пополнять нашу копилку знаний.

Цель

В этой главе мы продолжим и завершим разработку Twootr и параллельно рассмотрим ряд новых тем.

- Мы обсудим, как избежать связанности при помощи принципа инверсии зависимостей (DIP — Dependency Inversion Principle) и внедрения зависимости (DI — Dependency Injection).
- Мы займемся вопросом живучести и разберем шаблоны (паттерны) «Репозиторий» и «Объект-запрос».
- Мы рассмотрим короткое введение в функциональное программирование, которое даст представление об использовании идей из Java-ориентированного контекста в реальном приложении.

Резюме

Поскольку мы продолжаем проект, работу над которым начали в предыдущей главе, возможно, стоит пробежаться по основным моментам разработки. Если вы отлично помните код, можете спокойно пропустить этот раздел.

- `Twootr` — это родительский класс, определяющий основную логику и управляющий системой.
- `Twoot` — это одиночный экземпляр сообщения, публикуемого пользователем в системе.
- `ReceiverEndPoint` — это интерфейс, реализованный в UI-адаптере, который передает объекты `Twoot` в пользовательский интерфейс.
- `SenderEndPoint` содержит методы, взаимодействующие с событиями, которые пользователь отправляет в систему.
- Управление паролями и хэширование выполняется в классе `KeyGenerator`.

Живучесть и шаблон «Репозиторий»

На данный момент мы имеем систему, которая может поддерживать большинство операций твутинга. Однако если мы перезапустим процесс Java, то все твуты и вся пользовательская информация будут утеряны. Соответственно, нам нужен способ хранения информации, чтобы она не терялась при перезапуске. Ранее в дискуссии об архитектуре программы мы говорили о портах и адаптерах и о том, как сохранить ядро приложения независимым от системы хранения данных. Есть один весьма распространенный шаблон, который поможет нам в нашем вопросе: шаблон «Репозиторий».

Шаблон «*Репозиторий*» определяет интерфейс между доменной логикой приложения и системой хранения данных. Помимо того, что его применение позволит нам использовать различные системы хранения, такой подход имеет еще ряд преимуществ.

- Он обеспечивает централизацию логики обработки данных из системы хранения в доменную модель.
- Он позволяет проводить модульное тестирование логики без необходимости развертывания базы данных.
- Он повышает обслуживаемость и читаемость за счет сохранения единственной ответственности для каждого класса.

Репозиторий (хранилище) можно представить себе в виде коллекции объектов, но вместо простого складирования их в памяти репозиторий хранит их где-то в другом месте. При разработке нашего приложения мы вели разработку репозиториев через тестирование. Однако, чтобы не занимать время, здесь мы приведем только финальный результат. Поскольку репозиторий представляет собой коллекцию объектов, то в Twootr нам их нужно два: один для хранения объектов `User`, другой для объектов `Twoot`. У большинства репозиториев есть набор базовых операций:

`add()`

Записывает новый экземпляр объекта в репозиторий.

`get()`

Ищет одиночный объект по заданному идентификатору.

`delete()`

Удаляет экземпляр из хранилища.

`update()`

Проверяет, что значения, сохраненные для данного объекта, эквивалентны полям экземпляра.

Некоторые разработчики пользуются акронимом CRUD для описания данного набора операций. Он происходит от слов Создавать, Читать, Обновлять и Удалять (Create, Read, Update, Delete — CRUD). Однако мы заменили `create` и `read` на `add` и `get`, поскольку они больше соответствуют применяемой в Java терминологии, например если речь идет о работе с коллекциями.

Проектирование репозиториев

В нашем случае мы ведем разработку репозиториев через тесты и пользуемся нисходящим принципом («сверху вниз»). Особенность в том, что не все операции заданы для обоих репозиториев. В `UserRepository`, представленном в примере 7-1, нет операции удаления пользователя. Потому что нет соответствующего требования для реализации этой операции. Мы спросили об этом Джо и он сказал: «Твутнув однажды, ты уже не сможешь остановиться!»

Когда вы работаете сами, может возникнуть желание добавить еще функционал, чтобы иметь «нормальный» набор операций в репозитории, однако мы категорически против такого подхода. Неиспользуемый код, или, как еще говорят, *мертвый код*, — это помеха. В каком-то смысле любой

код — это помеха, однако если код действительно для чего-то нужен, то он хотя бы приносит пользу системе, в то время как неиспользуемый код — это просто помеха и больше ничего. По мере изменения требований к проекту нам придется модернизировать код. И чем больше неиспользуемого кода, тем сложнее становится эта задача.

Есть один принцип, на который мы ссылаемся в этой главе, но не упоминали ранее: YAGNI (от фразы «*Тебе это не понадобится*» — You ain't gonna need it). Это не значит, что не нужно применять абстракции и различные концепции вроде репозиториев. Просто не нужно писать код, который, как вы думаете, пригодится в будущем. Пишите его тогда, когда он вам необходим.

Пример 7-1. UserRepository

```
public interface UserRepository extends AutoCloseable {
    boolean add(User user);

    Optional<User> get(String userId);

    void update(User user);

    void clear();

    FollowStatus follow(User follower, User userToFollow);
}
```

Между нашими репозиториями есть разница, вызванная разными типами хранимых объектов. Объекты `Twoot` неизменяемые, так что `TwootRepository` из примера 7-2 не нужна реализация операции `update()`.

Пример 7-2. TwootRepository

```
public interface TwootRepository {
    Twoot add(String id, String userId, String content);

    Optional<Twoot> get(String id);

    void delete(Twoot twoot);

    void query(TwootQuery twootQuery, Consumer<Twoot> callback);

    void clear();
}
```

Обычно метод `add()` в репозитории просто берет объект и записывает его в базу данных. В случае с `TwootRepository` мы поступили иначе. Метод `add()` принимает определенные параметры и создает объект. Причина в том, что источник данных должен быть один, чтобы присвоить следующий объект `Position` к `Twoot`. Мы делегируем ответственность за обеспечение уникальности и упорядоченности объекта в уровень данных, у которого будут соответствующие инструменты для создания таких последовательностей.

Еще один вариант — взять объект `Twoot`, которому еще не присвоен `position`, и вносить поле `position` в момент добавления. Тогда одной из основных целей конструктора объекта будет проверка того, что все внутренние состояния полностью инициализированы, что хорошо достигается с полями `final`. Если не добавлять позицию в момент создания объекта, придется создавать объект, являющийся неполным экземпляром, что противоречит принципам создания объектов.

Некоторые реализации шаблона «Репозиторий» предлагают исходный интерфейс (пример 7-3). Однако нам это не очень подходит, потому что в `TwootRepository` нет метода `update()`, а в `UserRepository` нет метода `delete()`. Попытка избежать навязывания различных реализаций в один и тот же интерфейс является ключевой частью проектирования хорошей абстракции.

Пример 7-3. AbstractRepository

```
public interface AbstractRepository<T>
{
    void add(T value);

    Optional<T> get(String id);

    void update(T value);

    void delete(T value);
}
```

Объекты-запросы

Другая ключевая особенность репозиториев — поддержка запросов. В случае с Twootr репозиторию `UserRepository` не нужна поддержка запросов, но когда дело касается объектов `Twoot`, нам нужна возможность поиска твутов для повторного воспроизведения. Каким способом лучше реализовать данный функционал?

Что ж, есть несколько вариантов. Самый простой — попытаться организовать репозиторий как обычную коллекцию Java Collection и получить способ перебора различных объектов `Twoot`. В таком случае логику запросов можно будет написать в виде обычного Java-кода. Это прекрасное решение, однако код будет работать медленно, поскольку потребуется извлекать все строки данных из хранилища в приложение для того, чтобы сделать запрос, хотя на самом деле нам нужна только одна или несколько строк. Часто системы хранения и управления данными вроде баз данных SQL хорошо оптимизированы и имеют эффективные механизмы запросов и сортировки данных. Поэтому запросы лучше оставить им.

Осознавая, что реализация репозитория должна отвечать за запросы к данным, нам нужно решить, каким образом лучше организовать это в интерфейсе `TwootRepository`. Один из вариантов — создать метод, привязанный к основной логике приложения, который будет выполнять операцию запроса. К примеру, мы можем написать что-то вроде метода `twootsOnLogon()`, как показано в примере 7-4, который принимает объект пользователя и находит твуты, ассоциированные с ним. Недостаток такого подхода в том, что теперь появляется связь между конкретным функционалом логики с реализацией репозитория. Это усложняет последующую модернизацию, поскольку ведет к необходимости перерабатывать не только основную логику, но и алгоритмы репозитория, что противоречит принципу единственной ответственности.

Пример 7-4. `twootsForLogon`

```
List<Twoot> twootsForLogon(User user);
```

Мы хотим разработать что-то, что позволит нам сохранить возможность запросов к хранилищу данных без связанности системы управления данными и основной логики приложения. Мы можем создать специальный метод для запросов в репозиторий по заданным критериям, как показано в примере 7-5. Этот подход значительно лучше первых двух, однако стоит подумать еще чуть-чуть. Проблема в трудоемкости реализации каждого запроса в данном методе, ведь приложение будет со временем развиваться, и в него будет добавляться новый функционал запросов. А это значит, что количество методов в интерфейсе репозитория будет только расти, загромождая его и усложняя понимание.

Пример 7-5. `twootsFromUsersAfterPosition`

```
List<Twoot> twootsFromUsersAfterPosition(Set<String> inUsers, Position
lastSeenPosition);
```

Это подводит нас к варианту запроса, представленному в примере 7-6. Итак, мы абстрагировались от критерия, по которому делается запрос в TwootRepository. Теперь можно добавить дополнительные свойства этому критерию, чтобы делать запросы без необходимости множества методов, комбинирующихся в сложный запрос. Определение объекта TwootQuery показано в примере 7-7.

Пример 7-6. Запрос

```
List<Twoot> query(TwootQuery query);
```

Пример 7-7. TwootQuery

```
public class TwootQuery {
    private Set<String> inUsers;
    private Position lastSeenPosition;

    public Set<String> getInUsers() {
        return inUsers;
    }

    public Position getLastSeenPosition() {
        return lastSeenPosition;
    }

    public TwootQuery inUsers(final Set<String> inUsers) {
        this.inUsers = inUsers;

        return this;
    }

    public TwootQuery inUsers(String... inUsers) {
        return inUsers(new HashSet<>(Arrays.asList(inUsers)));
    }

    public TwootQuery lastSeenPosition(final Position lastSeenPosition) {
        this.lastSeenPosition = lastSeenPosition;

        return this;
    }

    public boolean hasUsers() {
        return inUsers != null && !inUsers.isEmpty();
    }
}
```

Стоит отметить, что это еще не последний вариант системы запросов для твутов. Возвращая объекты `List`, мы подразумеваем, что нам нужно загружать в память все объекты `Twoot`, возвращаемые за один раз. Это не очень хорошая идея, особенно если учитывать тот факт, что `List` может вырасти до весьма больших размеров. Мы не хотим запрашивать все объекты за один раз. Дело вот в чем: мы хотим выдавать пользовательскому интерфейсу каждый объект `Twoot` без необходимости хранить их все в памяти одновременно. Некоторые реализации репозиториев создают объект для моделирования набора возвращаемых результатов. Эти объекты позволяют нам «перелистывать» или «перебирать» все значения.

В нашем случае мы собираемся сделать кое-что попроще: просто воспользуемся обратным вызовом `Consumer<Twoot>`. Это функция, которую пользователь передает в виде аргумента, которая принимает один аргумент (`Twoot`) и возвращает `void`. Мы можем реализовать данный интерфейс при помощи лямбда-выражения или ссылки на метод. Наш последний вариант представлен в примере 7-8.

Пример 7-8. Запрос

```
void query(TwootQuery twootQuery, Consumer<Twoot> callback);
```

Обратитесь к примеру 7-9, чтобы посмотреть, как используется данный метод запроса. Вот как наш метод `onLogon()` вызывает запрос. Он берет авторизовавшегося пользователя и использует набор пользователей, на которых тот подписан, в этой части запроса. Обратный вызов, который принимает результаты данного запроса, — это `user::receiveTwoot`, ссылка на метод, который мы описывали ранее, передает объекты `Twoot` в `ReceiverEndPoint` пользовательского интерфейса.

Пример 7-9. Пример использования метода запроса

```
twootRepository.query(
    new TwootQuery()
        .inUsers(user.getFollowing())
        .lastSeenPosition(user.getLastSeenPosition()),
    user::receiveTwoot);
```

Вот наш интерфейс репозитория, разработанный и используемый в логическом ядре приложения.

Есть еще одна особенность, встречающаяся в других реализациях репозитория, о которой мы не рассказали. Речь идет о шаблоне *Unit of Work* («Единица работы»). Мы не используем этот шаблон в Twootr, однако он часто встречается в паре с шаблоном «Репозиторий», поэтому стоит о нем упомянуть. Общепринятый элемент линейных приложений — наличие операции, которая выполняет множество взаимодействий с хранилищем данных. Например, вы переводите деньги между двумя банковскими счетами и хотите забрать деньги с одного из них и поместить на другой, сделав это в рамках одной операции. При этом вы не хотите, чтобы из двух операций выполнилась только одна: вам не нужно класть деньги на второй счет, если на первом недостаточно средств. Также вы не хотите уменьшить баланс дебитора, не убедившись, что можете положить деньги на баланс кредитора.

Базы данных часто реализуют транзакции и совместимость ACID, чтобы позволить пользователям выполнять свои задачи. Транзакция — это набор различных операций с базой данных, которые логически представлены в виде отдельных, атомарных операций. Шаблон проектирования Unit of Work помогает в реализации транзакций с базой данных. Каждая выполняемая с репозиторием операция регистрируется объектом `unit of work`. Объект `unit of work` может затем делегировать в один или более репозиториев, объединяя эти операции в транзакцию.

Осталось еще кое-что, о чем мы не рассказали. Мы не упомянули о том, как мы реализуем наши интерфейсы репозитория. Опять же существует несколько возможных вариантов. В экосистеме Java есть много систем объектно-реляционного отображения (Object-Relational Mapping — ORM), предназначенных для решения задач, подобных нашей. Самая популярная система ORM — это Hibernate[1]. Системы ORM представляют собой простой способ автоматизации работы. Однако часто они являются источником не совсем оптимального кода для работы с базой данных и могут больше вносить сложности в программу, нежели нейтрализовать.

В примере проекта мы сделали две реализации каждого репозитория. Одна из них очень простая. Это реализация с хранением в памяти, подходящая для тестирования, но не сохраняющая данные при перезагрузке. Другая использует простые SQL и JDBC API. Мы не будем вдаваться в подробности этих реализаций, так как в большей части они не представляют особого интереса с точки зрения Java-программирования. Однако в следующем разделе мы поговорим о применении некоторых техник функционального программирования в данных реализациях.

[1] Подробнее на https://hibernate.org. — *Прим. авт.*

Функциональное программирование

Функциональное программирование — это стиль компьютерного программирования, в котором методы рассматриваются как математические функции. Это означает, что оно исключает изменяемое состояние и изменение данных. В таком стиле можно программировать на любом языке, однако в некоторых языках это делать проще и лучше. Такие языки мы называем *языками функционального программирования*. Java не относится к ним, однако в 8 версии, спустя 20 лет после первого релиза, начали появляться первые возможности функционального программирования. К этим возможностям относятся лямбда-выражения, потоки (Streams), Collectors API и класс Optional. В данном разделе мы немного поговорим о том, как можно использовать указанные возможности функционального программирования и о том, как применить их в Twootr.

До Java 8 у создателей библиотек были некоторые ограничения по уровню используемой абстракции. В качестве примера можно привести факт отсутствия эффективного механизма параллельных операций с большими коллекциями данных. Начиная с Java 8 у нас появилась возможность реализации сложных алгоритмов работы с коллекциями. Плюс при помощи простого вызова метода мы можем эффективно выполнять свой код на многоядерных процессорах. Однако для реализации набора библиотек для параллельной работы с данными Java потребовалось нововведние: лямбда-выражения.

Конечно, придется потратить время и силы, чтобы научиться писать и понимать «лямбда-ориентированный» код, однако, это того стоит. Проще изучить небольшой объем нового синтаксиса и пару новых выражений, чем вручную писать много сложного потокобезопасного кода. Хорошие библиотеки и платформы значительно снижают временные затраты на создание бизнес-приложений, поэтому любые барьеры в разработке простых и эффективных библиотек должны устраняться.

Абстракция — это принцип, знакомый любому, кто сталкивался с объектно-ориентированным программированием. Однако объектно-ориентированное программирование больше сконцентрировано на абстракции данных, в то время как функциональное программирование — на абстракции поведения. В реальности обе концепции нужны и важны, поэтому необходимо изучать их обе.

Упомянутая нами новая абстракция имеет свои плюсы. Тем из нас, кто не занимается на постоянной основе созданием кода, требовательного к производительности, она дает ряд очень важных преимуществ. Мы можем

писать более читаемый код — код, при создании которого мы вкладываем свои ресурсы в отображение цели своей логики, а не на механику ее достижения. Хорошо читаемый код является более удобным в обслуживании, более надежным и менее подвержен ошибкам.

Лямбда-выражения

Лямбда-выражение мы будем рассматривать как более простой способ описания анонимной функции. Мы понимаем, что для одного раза информации может быть многовато, поэтому попытаемся пояснить вам эту тему при помощи примеров из существующего Java-кода. Давайте начнем с интерфейса, используемого для представления обратного вызова — `ReceiverEndPoint` (пример 7-10).

Пример 7-10. `ReceiverEndPoint`

```
public interface ReceiverEndPoint {
    void onTwoot(Twoot twoot);
}
```

В данном примере мы создаем новый объект, который обеспечивает реализацию интерфейса `ReceiverEndPoint`. У этого интерфейса есть один метод — `onTwoot`, который вызывается объектом Twootr во время отправки им объекта `Twoot` в адаптер пользовательского интерфейса. Класс, показанный в примере 7-11, обеспечивает реализацию метода. В этом случае, чтобы не усложнять код, мы просто выводим его в командную строку вместо отправки данных в действующий интерфейс.

Пример 7-11. Реализация `ReceiverEndPoint` с классом

```
public class PrintingEndPoint implements ReceiverEndPoint {
    @Override
    public void onTwoot(final Twoot twoot) {
        System.out.println(twoot.getSenderId() + ": " + twoot.getContent());
    }
}
```

Мы рассмотрели пример параметризации поведения. Мы задаем параметры различным поведениям для отправки сообщения в пользовательский интерфейс.

Итак, мы видим семь строк стандартного кода, предназначенных, по сути, только для того, чтобы вызвать одну строку нужного нам действия. Анонимные классы были придуманы, чтобы упростить работу Java-программистов с действиями («поведениями»). Вы можете взглянуть на это в примере 7-12. Шаблонная часть кода несколько уменьшается, однако ее все еще нельзя считать достаточно легкой.

Пример 7-12. Реализация `ReceiverEndPoint` при помощи анонимного класса

```
final ReceiverEndPoint anonymousClass = new ReceiverEndPoint() {
    @Override
    public void onTwoot(final Twoot twoot) {
        System.out.println(twoot.getSenderId() + ": " + twoot.getContent());
    }
};
```

Стоит отметить, что шаблонный код — не единственная проблема. Такой код действительно тяжело читать, потому что он скрывает свое содержимое. Мы не хотим передавать объект. Все, что мы хотим — передавать некое поведение (исполняемый код). В Java 8 и более поздних версиях мы могли бы написать этот фрагмент кода в виде лямбда-выражения, как показано в примере 7-13.

Пример 7-13. Реализация `ReceiverEndPoint` при помощи лямбда-выражения

```
final ReceiverEndPoint lambda =
    twoot -> System.out.println(twoot.getSenderId() + ": " + twoot.getContent());
```

Вместо того чтобы передавать объект, который реализует интерфейс, мы передаем сразу блок кода — функцию без имени. `twoot` — это имя параметра, того же параметра, что и в примере с анонимным классом. Знак `->` разделяет параметр от тела лямбда-выражения, которое представляет собой участок кода, выполняемого в момент публикации твута.

Еще одно различие между этим примером и анонимным классом заключается в объявлении переменной события. Ранее нам нужно было явно указывать его тип: `Twoot twoot`. В этом примере мы не указывали тип вовсе, при этом пример компилируется. Что происходит под капотом? Компилятор `javac` выводит тип переменной из контекста. В конкретно нашем случае — из сигнатуры `onTwoot`. Это значит, что вам не нужно явно указывать тип, если он очевиден.

Несмотря на то что лямбда-выражения требуют гораздо меньше шаблонного кода, они все еще статически типизированы. В целях повышения читаемости у вас есть возможность включить объявления типов, однако компилятор не всегда может это обработать.

Ссылки на методы

Одно из самых распространенных выражений — это лямбда-выражение, вызывающее метод по его параметру. Если нам нужно лямбда-выражение, которое получает содержимое объекта `Twoot`, необходимо написать что-то вроде того, что показано в примере 7-14.

Пример 7-14. Получение содержимого твута

```
twoot -> twoot.getContent()
```

Данное выражение является настолько популярным, что для него даже существует сокращенный синтаксис, позволяющий повторно использовать существующий метод, и это называется ссылкой на метод. Если бы мы хотели переписать предыдущее лямбда-выражение с использованием ссылочного метода, то мы бы получили выражение, показанное в примере 7-15.

Пример 7-15. Ссылка на метод

```
Twoot::getContent
```

Стандартная форма выглядит как `Classname::methodName`. Помните, что, хотя это и метод, вам не нужны скобки, поскольку, по сути, вы не вызываете метод. Вы устанавливаете эквивалент лямбда-выражения, которое можно вызывать для вызова метода. Вы можете использовать ссылки на методы вместо лямбда-выражений, а также вызывать конструкторы при помощи описанного выше синтаксиса. Если бы вы использовали лямбда-выражение для создания `SenderEndPoint`, это выглядело бы так, как показано в примере 7-16.

Пример 7-16. Лямбда для создания нового `SenderEndPoint`

```
(user, twootr) -> new SenderEndPoint(user, twootr)
```

То же самое можно написать с использованием ссылок на методы (пример 7-17).

Пример 7-17. Ссылка на метод для создания `SenderEndPoint`

```
SenderEndPoint::new
```

Данный код не только короче, но и легче читается. Фрагмент `SenderEndPoint::new` незамедлительно говорит вам, что вы создаете новый объект `SenderEndPoint`, и освобождает вас от необходимости изучать всю строку кода. Кроме того, ссылки на методы автоматически поддерживают множественные параметры, пока у вас есть правильный функциональный интерфейс.

Когда мы знакомились с изменениями Java 8, наш друг сказал, что ссылки на методы «выглядят как обман». Он имел в виду, что, посмотрев на то, как мы можем использовать лямбда-выражения для передачи фрагментов кода так, как будто это данные, возможность ссылаться на метод напрямую — это обман.

На самом деле ссылки на методы действительно делают концепцию функций первого класса явной. Эта идея говорит о возможности передавать исполняемый код и пользоваться им как значением. К примеру, мы можем объединять функции.

Execute Around

Execute Around — очень распространенный шаблон в функциональном программировании. Вы можете столкнуться с ситуацией, когда у вас есть общий код инициализации и очистки, который всегда должен работать, но вы параметризуете различную логику, которая выполняется в коде инициализации и очистки. Пример типового шаблона показан на рис. 7-1. Есть несколько ситуаций, в которых вы можете использовать execute around, например:

Файлы

Откройте файл перед использованием, закройте его после использования. Также вы можете фиксировать исключения, если что-то идет не так. Параметризованный код способен читать из файла или записывать в него.

Блокировки

Воспользуйтесь блокировкой перед важным участком, снимите блокировку сразу после него. Параметризованный код — это важный участок.

Соединения базы данных

Откройте соединение с базой данных для инициализации, закройте соединение по завершении. Часто бывает полезнее объединить соединения с базой данных в пул, поскольку это позволяет вашей логике также извлекать соединение из пула.

Рис. 7-1. Шаблон Execute Around

Поскольку код инициализации и очистки используется во многих местах, можно попасть в ситуацию, когда он будет дублироваться. В таком случае при необходимости модификации кода инициализации или очистки вам придется изменять разные части приложения. Соответственно, возрастет риск того, что все эти разные фрагменты кода могут стать несовместимыми, то есть увеличивается вероятность возникновения багов.

Шаблон Execute Around решает эту проблему за счет общего метода, определяющего и код инициализации, и код очистки. Этот метод принимает параметр, содержащий исполняемый код (поведение), который отличается в зависимости от ситуаций использования. Параметр будет использовать интерфейс, чтобы позволить ему быть реализованным за счет различных блоков кода, обычно при помощи лямбда-выражений.

В примере 7-18 показан пример применения метода `extract`. Он используется в Twootr для выполнения выражений SQL базы данных. Здесь создается подготовленный объект для заданного выражения SQL, а затем запускается `extractor`. Метод `extractor` — это обратный вызов, который извлекает результат, то есть считывает данные из базы данных при помощи `PreparedStatement`.

Пример 7-18. Применение шаблона Execute Around в методе `extract`

```
    <R> R extract(final String sql, final Extractor<R> extractor) {
        try (var stmt = conn.prepareStatement(sql, Statement.RETURN_GENERATED_KEYS))
{
            stmt.clearParameters();
            return extractor.run(stmt);
```

```
        } catch (SQLException e) {
            throw new IllegalStateException(e);
        }
    }
```

Потоки

Наиболее важные особенности функционального программирования в Java сфокусированы на Collections API и *потоках* (Streams). Потоки позволяют нам писать код для работы с коллекциями с более высоким уровнем абстракции, чем позволяют циклы. Интерфейс `Stream` содержит набор функций, которые мы рассмотрим в этой главе. Каждая из них относится к общей операции, которую мы выполняли бы в `Collection`.

`map()`

Если у вас есть функция, которая преобразует значение одного типа в другой, `map()` позволяет вам применить функцию к потоку значений, генерируя новый поток обработанных значений.

Возможно, вы в течение многих лет вполне успешно делали операции конвертирования при помощи циклов. В `DatabaseTwootRepository` мы сделали кортеж, который используется в строке запроса и содержит все значения `id` различных пользователей, на которых подписан наш пользователь. Каждое значение `id` представляет собой фрагмент `String`, а весь кортеж заключен в скобки. Например, если у наших подписчиков идентификаторы `richardwarburto` и `raoul0K`, то мы получим `String`-кортеж `"(richardwarburto,raoul0K)"`. Для того чтобы сгенерировать такой кортеж, нужно использовать шаблон «мапинга» (mapping — сопоставление), преобразовывая каждый `id` в *id* и добавляя его в `List`. Метод `String.join()` можно использовать для объединения их через запятые. В примере 7-19 показан код, написанный в таком стиле.

Пример 7-19. Построение кортежа пользователей при помощи цикла

```
private String usersTupleLoop(final Set<String> following) {
    List<String> quotedIds = new ArrayList<>();
    for (String id : following) {
        quotedIds.add("'" + id + "'");
    }
    return '(' + String.join(",", quotedIds) + ')';
}
```

`map()` — одна из самых широко используемых операций в `Stream`. В примере 7-20 показан тот же код для кортежа пользователей, но уже с использованием функции `map()`.

Здесь также используются возможности `joining()`, что позволяет нам объединять элементы из потока в строку.

Пример 7-20. Построение кортежа пользователей при помощи `map`

```
private String usersTuple(final Set<String> following) {
    return following
        .stream()
        .map(id -> "'" + id + "'")
        .collect(Collectors.joining(",", "(", ")"));
}
```

Лямбда-выражение передается в `map()`, принимая `String` в качестве аргумента, и возвращает `String`. Не обязательно, чтобы аргумент и возвращаемое значение были одного типа, однако, переданное лямбда-выражение должно быть экземпляром `Function`. Это исходный функциональный интерфейс с одним аргументом.

`forEach()`

Операция `forEach()` удобна, когда вы хотите применить эффект для каждого значения в потоке. Например, вам нужно распечатывать имя пользователя или сохранять каждую транзакцию в потоке в базу данных. `forEach()` принимает один аргумент — обратный вызов `Consumer`, который вызывает каждый элемент потока в качестве аргумента.

`filter()`

Если вы циклически работаете над некоторыми данными и каждый элемент проверяете при помощи оператора `if`, вам стоит задуматься над использованием метода `Stream.filter()`.

Например, `InMemoryTwootRepository` нужно запрашивать различные объекты `Twoot`, чтобы найти объекты, соответствующие `TwootQuery`. Конкретно, если позиция больше, чем позиция последнего онлайн пользователя и если пользователь подписан. Образец, написанный в цикловом стиле, представлен в примере 7-21.

Пример 7-21. Обработка твутов в цикле при помощи оператора `if`

```
public void queryLoop(final TwootQuery twootQuery, final Consumer<Twoot> callback) {
    if (!twootQuery.hasUsers()) {
        return;
    }

    var lastSeenPosition = twootQuery.getLastSeenPosition();
    var inUsers = twootQuery.getInUsers();

    for (Twoot twoot : twoots) {
        if (inUsers.contains(twoot.getSenderId()) &&
            twoot.isAfter(lastSeenPosition)) {
            callback.accept(twoot);
        }
    }
}
```

Это называется шаблон «Фильтр». Основная идея фильтра — удерживать одни элементы потока, а другие пропускать. В примере 7-22 показано, как можно реализовать этот же код в функциональном стиле.

Пример 7-22. Функциональный стиль

```
@Override
public void query(final TwootQuery twootQuery, final Consumer<Twoot> callback) {
    if (!twootQuery.hasUsers()) {
        return;
    }

    var lastSeenPosition = twootQuery.getLastSeenPosition();
    var inUsers = twootQuery.getInUsers();

    twoots
        .stream()
        .filter(twoot -> inUsers.contains(twoot.getSenderId()))
        .filter(twoot -> twoot.isAfter(lastSeenPosition))
        .forEach(callback);
}
```

Также как и `map()`, `filter()` — это метод, который в качестве аргумента принимает одну функцию. В данном случае мы используем лямбда-выражение.

Эта функция делает ту же работу, что и функция `if` ранее. Здесь функция возвращает `true`, если строка начинается с цифры. Если вы работаете с унаследованным кодом, наличие оператора `if` внутри цикла `for` однозначно говорит о том, что вам нужен фильтр. Так как эта функция делает то же, что и оператор `if`, она должна вернуть или `true`, или `false` для данного значения. Поток после фильтра содержит значения из входного потока, который дает результат `true`.

reduce()

Этот шаблон знаком каждому, кто имел дело с циклами для работы с коллекциями. Вы применяете его, когда вам нужно обработать большой список значений. К примеру, найти сумму всех значений различных транзакций. Общий шаблон с применением цикла показан в примере 7-23. Используйте операцию `reduce`, если у вас есть коллекция значений, а на выходе вы хотите получить одно значение.

Пример 7-23. Шаблон reduce

```
Object accumulator = initialValue;
for (Object element : collection) {
 accumulator = combine(accumulator, element);
}
```

Объект `accumulator` «проталкивается» через тело цикла и выходит с итоговым значением, которое мы хотели вычислить. `accumulator` инициализируется с начальным значением `initValue`, а затем объединяется с каждым элементом списка за счет вызова операции `combine`.

Обратите внимание на элементы, которые отличаются в зависимости от реализации шаблона. Это `initialValue` и функция объединения. В исходном примере в качестве первого элемента списка мы использовали `initialValue`, но так не должно быть. Чтобы найти самое маленькое значение в списке, функция `combine` вернет кратчайший путь выхода из текущего элемента и `accumulator`. А теперь предлагаем посмотреть, как этот общий шаблон можно реализовать при помощи непосредственно операции в Streams API.

Давайте продемонстрируем операцию `reduce` путем добавления возможности комбинирования нескольких твутов в один большой твут. У этой операции будет список объектов `Twoot`, отправитель твута и его `id`, передаваемые

в виде аргументов. Операции нужно объединить различные по содержанию значения и вернуть наибольшую позицию объединенных твутов. Весь код представлен в примере 7-24.

Начнем с нового объекта `Twoot`, который создаем при помощи `id`, `senderId`, с пустым содержимым и с наименьшей возможной позицией — `INITIAL_POSITION`. Затем `reduce` складывает вместе каждый элемент с `accumulator`, добавляя по одному элементу на каждом шаге. Когда мы достигаем последнего элемента в потоке, в `accumulator` находится сумма всех элементов.

Лямбда-выражение, известное как уменьшитель (`reducer`), осуществляет объединение и принимает два аргумента. `acc` — это `accumulator`, в котором хранятся предыдущие твуты, которые были объединены. Он также передается в текущий `Twoot` в потоке. Уменьшитель в нашем примере создает новый `Twoot` с максимальной из двух позиций, конкатенацией их содержимого, и конкретными `id` и `senderId`.

Пример 7-24. Реализация суммы с использованием `reduce`

```
    private final BinaryOperator<Position> maxPosition = maxBy(comparingInt(Positio
n::getValue));

    Twoot combineTwootsBy(final List<Twoot> twoots, final String senderId, final
String newId) {
        return twoots
            .stream()
            .reduce(
                new Twoot(newId, senderId, "", INITIAL_POSITION),
                (acc, twoot) -> new Twoot(
                    newId,
                    senderId,
                    twoot.getContent() + acc.getContent(),
                    maxPosition.apply(acc.getPosition(), twoot.getPosition())));
    }
```

Естественно, сами по себе эти операции не так интересны. Они становятся по-настоящему мощным инструментом, когда мы объединяем их в «трубу». В примере 7-25 показан фрагмент кода из `Twootr.onSendTwoot()`, представляющий отправку твутов подписчикам пользователя. Первым делом мы вызываем метод `followers()`, который возвращает `Stream<User>`. Затем применяем операцию `filter`, чтобы найти пользователей, которые авторизованы

на данный момент и которым мы хотим отправить твут. После этого используем операцию forEach, чтобы получить желаемый эффект: отправить твут пользователям и записать результат.

Пример 7-25. Использование `Stream` в методе `onSendTwoot`

```
user.followers()
    .filter(User::isLoggedOn)
    .forEach(follower ->
    {
        follower.receiveTwoot(twoot);
        userRepository.update(follower);
    });
```

Optional

`Optional` — это тип данных из основной библиотеки Java, представленный в Java 8 и разработанный в качестве альтернативы `null`. К старому значению `null` есть много претензий. Даже человек, который изобрел эту концепцию, Тони Хоар, описывал ее как «свою ошибку на миллиард долларов»[1]. Быть одним из самых влиятельных людей в сфере компьютерных технологий, значит, иметь возможность совершить ошибку на миллиард долларов, даже не видя самого миллиарда.

`null` часто используется для представления отсутствия значения. И именно в такой ситуации `Optional` заменяет `null`. Проблема использования `null` в данном случае заключается в ужасной ошибке `NullPointerException`. Если вы ссылаетесь на переменную, которая имеет значение `null`, то ваш код аварийно завершает работу. У `Optional` двойная цель. Во-первых, побуждает программиста делать соответствующие проверки на отсутствующее значение, чтобы избежать багов. Во-вторых, документирует значения, которые ожидаются с отсутствующим значением в API класса. Это помогает найти скрытые баги.

Давайте взглянем на API для `Optional`, чтобы понять, как этим пользоваться. Если вы хотите создать экземпляр `Optional` из значения, есть рабочий метод под названием `of()`. Теперь Optional будет выступать в роли контейнера для этого значения, которое можно получить при помощи `get`, что показано в примере 7-26.

[1] Посмотреть видео выступления Тони Хоара «Null References: The Billion Dollar Mistake» можно на https://oreil.ly/OaXWj. — *Прим. авт.*

Пример 7-26. Создание `Optional` из значения

```
Optional<String> a = Optional.of("a");

assertEquals("a", a.get());
```

Поскольку `Optional` может также представлять и отсутствующее значение, есть рабочий метод под названием `empty()`, и вы можете конвертировать значение в `Optional` при помощи метода `ofNullable()`. Оба этих метода показаны в примере 7-27, также как и применение метода `isPresent()`, который показывает, содержится ли значение в `Optional`.

Пример 7-27. Создание пустого объекта `Optional` и проверка на наличие значения

```
Optional emptyOptional = Optional.empty();
Optional alsoEmpty = Optional.ofNullable(null);

assertFalse(emptyOptional.isPresent());

// a is defined above
assertTrue(a.isPresent());
```

Один из подходов к использованию `Optional` предусматривает защиту любого вызова `get()` за счет проверки `isPresent()`. Это необходимо, поскольку вызов `get()` может сгенерировать исключение `NoSuchElementException`. Как ни странно, такой подход — не лучший пример использования `Optional`. Если вы пользуетесь именно им, то можно сказать, что все, чем вы занимаетесь — это просто воспроизводите другие шаблоны для использования `null` (где вы в целях безопасности можете проверить, что значение не `null`).

Более изящным подходом является вызов метода `orElse()`, который выдает альтернативное значение, если контейнер `Optional` пуст. Если создание альтернативного значения требует много ресурсов, можно использовать метод `orElseGet()`. Это позволяет вам передавать функцию `Supplier`, которая вызывается только тогда, когда `Optional` действительно пуст. Оба эти метода демонстрируются в примере 7-28.

Пример 7-28. Использование `orElse()` и `orElseGet()`

```
assertEquals("b", emptyOptional.orElse("b"));
assertEquals("c", emptyOptional.orElseGet(() -> "c"));
```

В Optional также есть набор методов, которые могут использоваться как Stream API. Например, filter(), map(), ifPresent(). Применение этих методов в Optional API представляется похожим на Stream API, однако в данном случае ваш поток может содержать только 1 и 0 элементов. Поэтому Optional.filter() оставит элемент в Optional, если он соответствует критериям, и вернет пустой Optional, если он до этого был пуст или если предикат не соответствует действительности. Таким же образом map() преобразует внутреннее значение Optional, но если он пуст, то функция не применяется вообще. Поэтому применение этих функций безопаснее, чем null. Они взаимодействуют с Optional только если внутри него что-то есть. ifPresent это аналог forEach — он применяет обратный вызов Consumer, если присутствует значение, и никак иначе.

Вы можете увидеть метод Twootr.onLogon() в примере 7-29. Это пример того, как мы можем объединять различные операции для выполнения более сложной операции. Мы начинаем с поиска пользователя по идентификатору путем вызова метода UserRepository.get(), который возвращает Optional. Затем проверяем пользовательский пароль на соответствие при помощи filter. Мы используем ifPresent, чтобы уведомить пользователя о пропущенных твутах. Наконец, мы преобразовываем объект User в новый SenderEndPoint, который возвращается из метода.

Пример 7-29. Использование Optional в методе onLogon

```
var authenticatedUser = userRepository
    .get(userId)
    .filter(userOfSameId ->
    {
     var hashedPassword = KeyGenerator.hash(password, userOfSameId.getSalt());
        return Arrays.equals(hashedPassword, userOfSameId.getPassword());
    });

authenticatedUser.ifPresent(user ->
{
    user.onLogon(receiverEndPoint);
    twootRepository.query(
        new TwootQuery()
            .inUsers(user.getFollowing())
            .lastSeenPosition(user.getLastSeenPosition()),
        user::receiveTwoot);
    userRepository.update(user);
});

return authenticatedUser.map(user -> new SenderEndPoint(user, this));
```

В этом разделе мы увидели только верхушку функционального программирования. Если вам интересно более глубокое изучение функционального программирования, мы рекомендуем прочитать книги «*Современный язык Java*»[1] и «*Лямбда-выражения в Java 8*»[2].

Пользовательский интерфейс

На протяжении всей главы мы старались избегать разговоров о пользовательском интерфейсе, так как были сфокусированы на разработке ядра приложения. Теперь стоит немного углубиться в то, что образцовый проект представляет как часть своего пользовательского интерфейса, чтобы понять, как собрать вместе моделирование событий. Итак, наш проект — это одностраничный веб-сайт, в котором применяется JavaScript для реализации динамической функциональности. Чтобы сохранять все в достаточно простом виде и не погружаться в несметное число противоборствующих платформ, мы используем `jquery`, чтобы обновлять сырую HTML-страницу. Однако сохраняем простое разделение задач в коде.

Когда вы заходите на веб-страницу Twootr, он подключается к хосту при помощи WebSockets. Это один из тех способов связи, которые мы обсуждали в разделе «От событий к разработке». Весь код для связи находится в пакете `web_adapter chapter_06`. Класс `WebSocketEndPoint` реализует `ReceiverEndPoint`, а также вызывает любые нужные методы в `SenderEndPoint`. Например, когда получает и анализирует сообщение-запрос на подписку, он вызывает `SenderEndPoint.onFollow()`, передавая ему имя пользователя. Возвращаемое перечисление enum `FollowStatus` конвертируется в нужный формат для передачи и отправляется в соединение WebSocket.

Все взаимодействие между пользовательской частью JavaScript и сервером происходит при помощи стандарта JSON (*JavaScript Object Notation* — текстовый формат обмена данными)[3]. JSON был выбран потому, что в пользовательских интерфейсах JavaScript легко десериализовать и сериализовать.

В `WebSocketEndPoint` нам нужно преобразовывать в и из JSON при помощи Java. Есть много библиотек для этой цели. Мы выбрали наиболее популярную

[1] «Современный язык Java. Лямбда-выражения, потоки и функциональное программирование», Рауль-Габриэль Урма, Алан Майкрофт, Марио Фуско. — *Прим. ред.*

[2] «Лямбда-выражения в Java 8. Функциональное программирование — в массы», Ричард Уорбертон. — *Прим. ред.*

[3] Подробнее на https://www.json.org. — *Прим. авт.*

библиотеку Jackson, которая хорошо поддерживается. JSON часто используется в приложениях, в которых применяется подход запрос/ответ вместо событийно-ориентированного подхода. В нашем случае мы вручную извлекаем поля из объекта JSON, чтобы не усложнять конструкцию. Однако возможно использовать более высокоуровневый JSON API вроде Binding API.

Инверсия зависимости и внедрение зависимости

В этой главе мы много говорили о шаблонах развязывания. Наше приложение использует шаблон портов и адаптеров и шаблон репозитория, чтобы развязать бизнес-логику и детали реализации. Есть большой унифицирующий принцип, о котором мы думаем, когда сталкиваемся с этими шаблонами — *Принцип инверсии зависимости* (DIP — Dependency Inversion Principle), последний из пяти шаблонов SOLID, рассматриваемых в этой книге. Как и все остальные, он был введен Робертом Мартином. Данный принцип утверждает, что:

- Модули высокого уровня не должны зависеть от модулей низкого уровня. Оба уровня должны зависеть от абстракций.
- Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

Принцип называется принципом инверсии потому, что традиционно в структурном программировании встречаются случаи, когда модули высокого уровня производят модули низкого уровня. Часто это является побочным эффектом нисходящей технологии разработки, о которой мы говорили ранее. Большую проблему вы разбиваете на подпроблемы, создаете модули для их решения, и тогда главная проблема (модуль высокого уровня) зависит от подпроблем (модулей низкого уровня).

При разработке Twootr нам удалось избежать таких проблем за счет абстракций. У нас есть высокоуровневый входной класс Twootr, и он не зависит от модулей низкого уровня вроде `DataUserRepository`. Он зависит от абстракции — интерфейса `UserRepository`. Такую же инверсию мы производим в порте пользовательского интерфейса. Twootr не зависит от `WebSocketEndPoint`, он зависит от `ReceiverEndPoint`. Программа сводится к интерфейсу, а не к реализации.

Связанное понятие — *концепция внедрения зависимости* (DI — Dependency Injection). Чтобы понять, что такое концепция DI и зачем она нужна, давайте мысленно проведем эксперимент. Итак, было принято решение, что

главный класс Twootr должен зависеть от `UserRepository` и `TwootRepository`, чтобы хранить объекты `User` и `Twoot`. Внутри Twootr мы определили поля для хранения экземпляров этих объектов, как показано в примере 7-30. Вопрос в том, как нам создать эти экземпляры?

Пример 7-30. Зависимости в классе `Twootr`

```
public class Twootr
{
    private final TwootRepository twootRepository;
    private final UserRepository userRepository;
```

Первый способ — вызывать конструкторы с помощью ключевого слова `new`, как показано в примере 7-31. Здесь мы жестко запрограммировали использование репозиториев. Большая часть кода в классе все еще нацелена на интерфейс, поэтому нам нужно немного изменить реализацию, но это не совсем честно. Мы должны всегда использовать репозитории без данных, что означает, что наши тесты класса Twootr зависят от базы данных и, соответственно, работают медленнее.

Кроме того, если мы будем поставлять различные версии Twootr различным клиентам, к примеру, «стационарную» версию Twootr, использующую SQL, для компаний-клиентов и облачную версию, использующую NoSQL, то нам придется отделять сборки для двух различных версий кода. Недостаточно просто определить интерфейсы и разделить реализации: нам также нужен способ связи с правильной реализацией, который не будет разрушать нашу абстракцию и нарушать подход развязывания.

Пример 7-31. Жестко запрограммированные экземпляры полей

```
public Twootr()
{
    this.userRepository = new DatabaseUserRepository();
    this.twootRepository = new DatabaseTwootRepository();
}

// How to start Twootr
Twootr twootr = new Twootr();
```

Наиболее распространенным шаблоном для осуществления различных зависимостей является шаблон Abstract Factory (Абстрактная фабрика). Он

показан в примере 7-32. Здесь у нас есть рабочий («фабричный») метод, который мы можем использовать для создания экземпляра нашего интерфейса при помощи метода `getInstance()`. Когда мы хотим настроить нужную реализацию, мы можем вызывать `setInstance()`. Так, например, мы можем использовать `setInstance()` в тестах, чтобы создать реализацию в памяти, в локальной установке для использования базы данных SQL или в облачной среде для использования базы данных NoSQL. Мы отвязали реализацию от интерфейса и можем вызывать этот код откуда угодно.

Пример 7-32. Создание экземпляров при помощи рабочих методов

```
public Twootr()
{
    this.userRepository = UserRepository.getInstance();
    this.twootRepository = TwootRepository.getInstance();
}

// How to start Twootr
UserRepository.setInstance(new DatabaseUserRepository());
TwootRepository.setInstance(new DatabaseTwootRepository());
Twootr twootr = new Twootr();
```

Как ни странно, у фабричного подхода есть свои недостатки. Для начала мы создали «большой комок» общего изменяемого состояния. В любой ситуации, когда нам нужно запустить одну JVM с различными экземплярами Twootr с разными зависимостями, это будет невозможно. Кроме того, мы связали жизненные циклы. Возможно, когда-то нам захочется создать новый экземпляр `TwootRepository` при запуске Twootr или использовать уже существующий. Фабричный подход не позволит нам сделать это напрямую. Помимо этого, было бы гораздо сложнее иметь фабричный метод для каждой зависимости, которую мы хотим реализовать в приложении.

Настал момент для появления внедрения зависимости. Можно представить, что DI это «агент-посредник» — не звоните нам, мы сами вам позвоним. С помощью DI вместо явного создания зависимостей или использования фабрик для их создания мы просто берем параметр, и любой экземпляр нашего объекта несет ответственность за передачу требуемых зависимостей. Им может быть метод настройки тестового класса, который передается в мок-объект. Или метод `main()` нашего приложения в реализации базы данных SQL (пример 7-33). Инверсия зависимости — это стратегия. Внедрение зависимости и шаблон репозитория — это тактики.

Пример 7-33. Создание экземпляров с использованием внедрения зависимости

```
public Twootr(final UserRepository userRepository, final TwootRepository twootRepos
tory)
{
    this.userRepository = userRepository;
    this.twootRepository = twootRepository;
}

// How to start Twootr
Twootr twootr = new Twootr(new DatabaseUserRepository(), new DatabaseTwootRepos
tory());
```

Получение объектов таким способом не только упрощает создание тестов для них, но и имеет преимущество в экстернализации (вынесении) создания самих объектов. Это позволяет коду нашего приложения контролировать, в какой момент создается `UserRepository` и какие зависимости в него заложены. Многие разработчики считают удобным применение DI-фреймворков, таких как Spring и Guice, предлагающих множество возможностей по сравнению с базовым DI. Например, они задают жизненные циклы для bean-объектов, которые стандартизируют хуки, вызываемые после создания экземпляра объектов или до их уничтожения, если это необходимо. Они также могут предлагать области для объектов (вроде объектов Singleton), которые создаются только один раз в течение жизненного цикла процесса или объектов по запросу. Забегая вперед, эти DI-платформы часто хорошо интегрируются с платформами веб-разработки, такими как Dropwizard или Spring Boot, обеспечивая продуктивную работу.

Пакеты и сборочные системы

Java позволяет разделить код на различные пакеты. В этой книге код каждой главы мы помещали в отдельном пакете. Twootr — первый проект, при работе над которым мы разделили проект на несколько подпакетов.

Вот пакеты, в которых размещены различные компоненты этого проекта:

- `com.iteratrlearning.shu_book.chapter_06` — пакет верхнего уровня в проекте.
- `com.iteratrlearning.shu_book.chapter_06.database` — содержит адаптер для базы данных SQL.

- `com.iteratrlearning.shu_book.chapter_06.in_memory` — содержит адаптер для хранения в памяти.
- `com.iteratrlearning.shu_book.chapter_06.web_adapter` — содержит адаптер для пользовательского интерфейса, основанного на WebSockets.

Разделение больших проектов на отдельные пакеты помогает структурировать код и упрощает разработчикам поиск. Так же, как классы группируют методы и состояния, пакеты группируют связанные классы. Пакеты должны подчиняться тем же правилам связности и связанности, что и классы. Помещайте классы в один пакет, если они должны изменяться одновременно и если они относятся к одной и той же структуре. Например, в проекте Twootr, если мы хотим модернизировать код хранилища SQL, то знаем, что должны обратиться к подпакету `database`.

Пакеты, кроме всего прочего, позволяют скрывать информацию. При рассмотрении примера 4-3 мы обсуждали идею пакетного конструктора, чтобы исключить возможность создания экземпляров объектов вне пакета.

Мы также можем ввести «пакетность» для классов и методов. Это исключит возможность доступа внешних объектов к деталям класса и поможет достичь снижения связанности. Например, `WebSocketEndPoint` — это пакетная реализация интерфейса `ReceiverEndPoint`, которая находится в пакете `web_adapter`. Никакой другой код в проекте не может обращаться к этому классу напрямую. Только через интерфейс `ReceiverEndPoint`, выполняющий роль порта.

Наша идея о своем пакете для каждого адаптера хорошо перекликается с шаблоном гексагональной архитектуры, который мы задействовали в проекте. Стоит отметить, что не каждое приложение является гексагональным. Поэтому есть две распространенные структуры пакетов, с которыми вы можете столкнуться в других проектах.

Один из распространенных способов структурирования пакетов — структурирование по слоям. Например, сгруппировать весь код, ответственный за HTML-страницу, в пакет `views`, а весь код, который обрабатывает вебзапросы, — в пакет `controller`. Несмотря на свою популярность, такой подход к структурированию не всегда является удачным, потому что приводит к плохой связности и связанности. Если вы захотите поменять свою вебстраницу, добавив туда какой-то параметр, и выводить значение, основанное на этом параметре, то вам придется иметь дело и с пакетом `controller`, и с пакетом `views`, а возможно, и с какими-то еще.

Альтернативный способ группировки пакетов — группировка по свойствам. Например, если вы создаете сайт для электронной коммерции, у вас будет пакет `cart` для покупательской корзины, пакет `product` для кода, связанного с перечнем товаров, пакет `payment` для кода, связанного с платежными картами, и т. д. Такой вариант обеспечит лучшую связность. Если вы, уже осуществляя поддержку карт Visa, вдруг захотите реализовать поддержку еще и системы Mastercard, то вам нужно будет модернизировать только пакет `payment`.

В разделе «Работа с Maven» мы говорили о том, как настроить базовую структуру сборки при помощи сборщика Maven. Если рассматривать как структуру проекта структуру данной книги, то мы имеем один проект Maven, а различные главы книги — это различные Java-пакеты одного проекта. Это хорошая и простая структура, которая подойдет к большинству проектов. Однако она не единственная. И Maven, и Gradle предлагают структуры проектов, которые создают и выводят множество артефактов (элементов) сборки из одного проекта верхнего уровня.

Это может быть полезно, если вы хотите развернуть различные сборочные артефакты. Предположим, у вас есть клиент-серверный проект и вам нужно получить одну сборку на выходе, в которой будут и клиентская, и серверная части, в то время, как клиент и сервер — это разные бинарные элементы, работающие на разных машинах. Однако не стоит слишком усложнять сборочные скрипты.

Вы и ваши коллеги будете часто запускать их на своих машинах, поэтому наивысший приоритет здесь — простота в использовании и скорость. Вот почему мы идем по пути общего проекта для всей книги, вместо отдельных подмодулей для каждого проекта.

Ограничения и упрощения

Вы познакомились с нашей реализацией Twootr и изучили соответствующие проектные решения. Но означает ли это, что данный код Twootr является единственно правильным? Конечно нет! На самом деле в нашем подходе есть некоторые ограничения и упрощения, которые мы преднамеренно ввели, чтобы сохранить возможность рассмотреть весь код в рамках одной главы.

Прежде всего, мы написали Twootr для работы в одном потоке, полностью игнорируя вопросы конкурентности. На практике мы реализовали бы

многопоточность для взаимодействия с событиями в Twootr. Таким образом мы смогли бы использовать возможности современных процессоров и обслуживать большее количество пользователей одной машиной.

Кроме того, мы полностью проигнорировали вопрос отказоустойчивости, касающийся работы сервиса в случае отказа сервера. Мы также не уделили внимания масштабируемости. Например, запрос всех твутов происходит определенным порядком, который легко реализуется на одном сервере, но может стать весьма узким местом в системе. Точно так же может привести к «затору» и просмотр всех твутов сразу. Представьте, если вы уедете в отпуск на неделю и по возвращении вас будет ждать 20 000 твутов!

Рассмотрение данных проблем в деталях не вписывается в рамки нашей главы. Однако если вы планируете продолжать изучать Java, эти важные вопросы стоит рассмотреть. И мы планируем заняться ими в наших следующих книгах.

Выводы

- Теперь вы можете отвязать хранилище данных от бизнес-логики приложения при помощи шаблона репозитория.
- Вы познакомились с реализациями двух типов репозиториев с таким подходом.
- Вам были представлены идеи функционального программирования, включая потоки в Java 8.
- Вы узнали, как структурировать большой проект при помощи разных пакетов.

Самостоятельная работа

Если вы хотите расширить и укрепить полученные в этой главе знания, попробуйте выполнить что-нибудь из перечисленных ниже заданий.

Предположим, что для Twootr мы выбрали технологию pull. Вместо постоянного «проталкивания» сообщений к браузерному клиенту через WebSockets мы используем HTTP для запроса последних не просмотренных сообщений.

- Подумайте, как можно изменить нашу разработку. Попробуйте нарисовать схему взаимодействия между классами и отразить, как между классами передаются данные.
- При помощи TDD реализуйте альтернативную модель Twootr. Вам не нужно реализовывать HTTP часть. Только основные классы для этой модели.

В завершение

Мы создали продукт, и он работает. Неожиданно Джо понял, что некто по имени Джек уже представил похожий продукт с похожим названием, имеет сотни миллионов пользователей и зарабатывает миллиарды. Джек сделал это первым несколько лет назад. Конечно, Джо очень расстроился.

Заключение

Если вы дочитали до этих строк, значит, мы надеемся, книга вам понравилась. Что касается нас, то мы получили удовольствие при ее написании. В этой заключительной главе мы расскажем вам, куда двигаться дальше. Мы дадим несколько советов по развитию навыков и повышению вашего уровня.

Проектно-ориентированная структура

Проектно-ориентированная структура книги была придумана для того, чтобы помочь вам в понимании принципов разработки. Мы предлагали вам темы совместно с программными проектами, чтобы сформировать понимание принятия решений в контексте разработки программного обеспечения. Контекст — это критически важный элемент в разработке. Одно решение может быть правильным в одном контексте и совершенно неприемлемым в другом. Многие разработчики злоупотребляют подклассами ввиду недостаточного понимания того, что это механизм повторного использования кода. Надеемся, что в главе 4 мы разоблачили эту идею.

Стоит отметить, что просто прочтение книги не сделает вас экспертом в разработке программного обеспечения. Для этого нужны практика, опыт и терпение. Данная книга способна только оптимизировать и улучшить этот процесс. Вот зачем мы ввели разделы «Самостоятельная работа» в каждой главе. Они способствуют пониманию и закреплению материала книги.

Самостоятельная работа

Как разработчик вы, вероятно, часто приступаете к проектам итеративным способом. Другими словами, уделяете неделю или две реализации наиболее приоритетных элементов, а затем при помощи обратной связи определяете

следующий набор элементов. Мы обнаружили, что имеет смысл оценивать прогресс в навыках таким же образом. Если периодически «оглядываться назад», это поможет вам сфокусироваться и определить направление для работы. Интенсивная разработка часто подразумевает еженедельную ретроспективу, но лично вам не нужно делать это так часто. Ретроспектива раз в квартал или раз в два года была бы весьма кстати. Еще один полезный момент — оценивать, какие навыки пригодятся вам в текущей или будущей работе. Чтобы убедиться, что данные навыки развиваются, стоит ставить цели на квартал. Они должны учитывать все, что вы хотите изучить или улучшить. Не обязательно ставить какую-то большую цель вроде изучения нового языка программирования. Лучше сосредоточиться на чем-то несложном. Например, освоить новую тестовую платформу или пару шаблонов проектирования.

Что касается новых навыков, мы часто слышим такой вопрос от разработчиков: «Как я могу постоянно учиться новым технологиям, методикам и принципам?» Это нелегко, ведь каждый занят своими делами. Не стоит пытаться изучить все. Это верный путь к сумасшествию! Способность определять нужные навыки, которые будут служить вам в течение долгого времени, — вот что делает из вас хорошего разработчика. Ключевой момент — всегда развиваться и работать над собой.

Сознательная практика

Хотя данная книга и покрывает много ключевых вопросов и навыков, необходимых для хорошего разработчика, очень важно практиковать их. Чтения самого по себе недостаточно. Только практика поможет вам усвоить новые навыки. Стремитесь в своей повседневной работе всегда искать оптимальные решения. Каждый шаблон, описанный в этой книге, где-то можно применять, а где-то не стоит. Мы постарались показать вам, как важно заранее оценивать, в каких ситуациях рассматриваемая вами методика выигрывает, а в каких — нет.

Часто мы думаем, что природный талант и интеллект — наиболее определяющие факторы успеха, однако большинство исследований показывает, что практика и работа — истинные ключи к успеху. Такие книги, как «*Выдающиеся результаты. Талант ни при чем!*» Джеффа Колвина и «*Гении и аутсайдеры. Почему одним все, а другим ничего?*» Малкольма Гладуэлла, оценивают ряд ключевых факторов успеха в жизни, и наиболее эффективным из них является сознательная практика.

Сознательная практика — это разновидность практики, являющейся систематической и имеющей цель. Сознательная практика помогает улучшить производительность, что требует концентрации внимания. Часто, когда люди практикуют свои навыки, чтобы усовершенствовать их, они просто занимаются повторением. Делать одно и то же снова и снова, ожидая улучшений — это не самый эффективный способ.

Приведем пример. Когда мы изучали библиотеку Eclipse Collections[1], мы, стремясь освоить и понять ее системным путем, прошли через замечательный набор упражнений, поставляющихся с данной библиотекой. Чтобы убедиться, что мы действительно хорошо усвоили материал, мы выполнили все упражнения три раза. Каждый раз мы сравнивали свои результаты с предыдущими, находя более четкие и быстрые способы решения.

Повторение собственных действий приводит к автоматизму. Поэтому если в процессе работы вы приобрели нехорошую привычку, вы можете приучить себя к ней, постоянно применяя в работе. Опыт укрепляет привычку. Сознательная практика — хороший способ разорвать этот цикл. Она помогает системно внедрять новые подходы. Например, можно решать какую-то небольшую задачу, каждый раз применяя новый подход. Это можно организовать самостоятельно, либо на специальных учебных курсах. Неважно, каким именно путем вы пойдете. Сознательная практика — это ключ к оттачиванию ваших навыков и применению новых знаний, в том числе полученных из этой книги.

Следующие шаги и дополнительные ресурсы

Что ж, надеемся, вы понимаете, что эта книга — не конец пути. Но чего искать дальше?

Отличный способ получить новые знания и расширить горизонты — погрузиться в ПО с открытым исходным кодом. Большинство таких Java-проектов, например, JUnit и Spring, находятся на GitHub[2]. Какие-то из этих проектов могут быть более «дружелюбными», чем остальные. Часто разработчики открытых проектов очень заняты и им нужна помощь. Советуем ознакомиться со списком ошибок и посмотреть, можете ли вы им чем-то помочь.

---

[1] https://www.eclipse.org/collectionswww.eclipse.org/collections/. — *Прим. авт.*
[2] https://github.com. — *Прим. авт.*

Не стоит забывать и про классические курсы и онлайн-обучение — еще один распространенный способ развития навыков. Популярность онлайн-курсов постоянно растет. Можем сказать, что на Pluralsight[1] и на O'Reilly Learning Platform[2] есть отличные варианты курсов Java.

Другой крутой источник информации для разработчиков — это блоги и Twitter. И Ричард[3], и Рауль[4] есть в Twitter и часто публикуют интересные ссылки. Programming Reddit[5] часто выступает в качестве агрегатора ссылок, как и Hacker News[6]. Наконец, учебная компания, которой управляют авторы книги, тоже предлагает серию бесплатных статей[7].

Спасибо, что прочитали нашу книгу. Мы ценим ваши мысли и отзывы и желаем вам успехов в Java-разработке.

---

[1] https://www.pluralsight.com. — *Прим. авт.*
[2] https://www.oreilly.com. — *Прим. авт.*
[3] https://twitter.com/richardwarburto. — *Прим. авт.*
[4] https://twitter.com/raouluk. — *Прим. авт.*
[5] https://www.reddit.com/r/programming. — *Прим. авт.*
[6] https://news.ycombinator.com. — *Прим. авт.*
[7] http://iteratrlearning.com/articles. — *Прим. авт.*

Об авторах

Доктор Рауль-Габриэль Урма — исполнительный директор и основатель Cambridge Spark, ведущей обучающей организации в сфере ИТ, искусственного интеллекта, карьерного роста и продвижения. Является автором нескольких книг по программированию, включая такой бестселлер, как «Современный язык Java» (Modern Java in Action). Рауль-Габриэль имеет степень доктора философии по информатике Кембриджского университета, а также степень магистра Лондонского Имперского колледжа. Он окончил учебу с отличием первой степени, получил несколько наград в области технических инноваций. Его научные интересы связаны с языками программирования, компиляторами, анализом исходных кодов, машинным обучением и образованием. Был номинирован в качестве Java-чемпиона Oracle в 2017 году. Также является опытным международным оратором, проводит лекции по Java, Python, искусственному интеллекту и бизнесу. Рауль консультировал и работал с такими компаниями, как Google, Oracle, eBay и Goldman Sachs.

Доктор Ричард Уорбэртон — основатель Opsian.com, мэнтейнер Artio FIX Engine. Работал в качестве разработчика в различных сферах, в том числе занимался разработкой инструментов, HFT (высокочастотный трейдинг) и сетевых протоколов. Автор успешной книги «Лямбда-выражения в Java 8» (Java 8 Lambdas). Занимается обучением разработчиков на ресурсах http://iteratrlearning.com и https://www.pluralsight.com/authors/richard-warburton. Ричард — опытный оратор, он провел десятки встреч, регулярно выступает в качестве организатора на крупнейших конференциях Европы и США. Имеет степень доктора философии в сфере информатики Уорикского университета.

В завершение

Животное на обложке книги — это красноголовый мангабей (Cercocebus torquatus), обезьяна Старого Света, найденная в горном массиве вдоль западного побережья Африки. Мангабеи обитают в лесной среде: как в болотах, так и на равнинах. Большую часть времени проводят на деревьях, забираясь на высоту до 100 футов (около 30 метров), спускаясь на землю для поиска пищи (особенно в сухой период). Питаются фруктами, семенами, орехами, растениями, грибами, насекомыми и птичьими яйцами.

Красноголовый мангабей получил свое название из-за белого воротника, выделяющегося на фоне более темного тела, а также из-за каштаново-красной головы. Белые веки подчеркивают и без того выразительную мордочку. Представители этого вида весят в среднем 20–22 фунта (около 9–10 кг) и имеют рост 18–24 дюйма (45–60 см). Как и многие древесные приматы, мангабей обладает длинным гибким хвостом, длиннее своего тела. Латинское название Cercocebus фактически означает «хвост обезьяны».

Мангабеи живут большими группами от 10 до 35 особей, состоящими из альфа-самца и самок с детенышами. Взрослые самцы живут в одиночестве до тех пор, пока не смогут сформировать или найти отряд (название для группы мангабеев), чтобы возглавить его. Оснащенные большими усилительными горловыми мешками, эти животные очень «вокальны», располагают большим репертуаром криков, хрюканья, кудахтанья и других звуков, которые служат для оповещения стада о хищниках или для предупреждения о вторжении злоумышленника. К сожалению, количество шума, производимого мангабеями, также делает их легкой мишенью для охотников, добывающих мясо диких животных. Мангабеи занесены в Красную книгу.

Предметный указатель

В

Г

Д

З

И

П

Р

С

Т

У

Ф

Х

Ц

Ч

Ш

Э

Я

Все права защищены. Книга или любая ее часть не может быть скопирована, воспроизведена в электронной или механической форме, в виде фотокопии, записи в память ЭВМ, репродукции или каким-либо иным способом, а также использована в любой информационной системе без получения разрешения от издателя. Копирование, воспроизведение и иное использование книги или ее части без согласия издателя является незаконным и влечет уголовную, административную и гражданскую ответственность

Научно-популярное издание

МИРОВОЙ КОМПЬЮТЕРНЫЙ БЕСТСЕЛЛЕР

Рауль-Габриэль Урма
Ричард Уорбертон

ГИД JAVA-РАЗРАБОТЧИКА
ПРОЕКТНО-ОРИЕНТИРОВАННЫЙ ПОДХОД

Главный редактор *Р Фасхутдинов*
Руководитель направления *В. Обручев*
Ответственный редактор *Е Истомина*
Литературный редактор *А Голанцева*
Младший редактор *А. Захарова*
Художественный редактор *А Гусев*
Компьютерная верстка *Э Брегис*
Корректоры *Л. Макарова, А Баскакова*

Страна происхождения Российская Федерация
Шығарылған елі: Ресей Федерациясы

ООО «Издательство «Эксмо»
123308, Россия, город Москва, улица Зорге, дом 1, строение 1, этаж 20, каб 2013
Тел 8 (495) 411 68 86
Home page www.eksmo.ru E-mail info@eksmo.ru
Өндіруші «ЭКСМО» АҚБ Баспасы,
123308 Ресей, қала Мәскеу, Зорге көшесі 1 үй, 1 ғимарат 20 қабат офис 2013 ж
Тел 8 (495) 411 68 86
Home page www.eksmo.ru E mail info@eksmo.ru
Тауар белгісі «Эксмо»
Интернет-магазин www.book24.ru

Интернет-магазин www.book24.kz
Интернет-дукен www.book24.kz
Импортёр в Республику Казахстан ТОО «РДЦ Алматы»
Қазақстан Республикасындағы импорттаушы «РДЦ-Алматы» ЖШС
Дистрибьютор и представитель по приему претензий на продукцию
в Республике Казахстан ТОО «РДЦ Алматы»
Қазақстан Республикасында дистрибьютор және өнім бойынша арыз-талаптарды
қабылдаушының өкілі «РДЦ-Алматы» ЖШС
Алматы қ, Домбровский көш., 3-а» литер Б, офис 1
Тел 8 (727) 251-59 90/91/92 E mail RDC-Almaty@eksmo.kz
Өнімнің жарамдылық мерзімі шектелмеген
Сертификация туралы ақпарат сайтта www.eksmo.ru/certification

Сведения о подтверждении соответствия издания согласно законодательству РФ
о техническом регулировании можно получить на сайте Издательства «Эксмо»
www.eksmo.ru/certification
Өндірген мемлекет Ресей Сертификация қарастырылмаған

Дата изготовления / Подписано в печать 11.10 2021.
Формат 70x100$^{1}/_{16}$. Печать офсетная. Усл печ л 18,15
Тираж 2000 экз Заказ № 9738

Отпечатано в АО «Можайский полиграфический комбинат»
143200, Россия, г Можайск, ул Мира, 93
www oaompk ru, тел (495) 748-04-67, (49638) 20-685

ЧИТАЙ ГОРОД

В электронном виде ... www.litres.ru ЛитРес:

ПРИСОЕДИНЯЙТЕСЬ К НАМ!

БОМБОРА
ИЗДАТЕЛЬСТВО

БОМБОРА – лидер на рынке полезных и вдохновляющих книг. Мы любим книги и создаем их, чтобы вы могли творить, открывать мир, пробовать новое, расти Быть счастливыми Быть на волне

МЫ В СОЦСЕТЯХ:

bomborabooks bombora
bombora.ru

ISBN 978-5-04-094955-7

12+

Официальный интернет магазин издательской группы "ЭКСМО-АСТ"

book24.ru

O'REILLY®

Гид Java-разработчика

Освоение множества современных концепций разработки программного обеспечения — утомительное занятие, особенно если вы только начинаете свою карьеру в Java. Нужно ли изучать приемы объектно-ориентированного программирования, такие, например, как разработка через тестирование?

Или стоит применять идеи функционального программирования? В этом практическом руководстве содержится интегрированный проектно-ориентированный подход, который поможет вам освоить ключевые навыки, необходимые, чтобы стать эффективным разработчиком.

Авторы демонстрируют свой подход на примере реальных проектов, начинающихся как простые консольные приложения и вырастающих в полноценные приложения. Если вы знаете основы программирования на Java, то здесь вы изучите современные методы разработки программного обеспечения, позволяющие создавать актуальные, стабильные и легкие в обслуживании Java приложения.

- **Изучите ключевые принципы разработки поддерживаемого кода.**
- **Сделайте исходники более гибкими, а код — поддерживаемым.**
- **Поймите, как применять связанность, связность и принципы SOLID.**
- **Используйте разработку через тестирование.**
- **Смотрите со стороны: двигайтесь от общей картины к ядру приложения.**
- **Познакомьтесь с основами функционального программирования и научитесь применять их в Java.**

«Эта книга заполняет пустоту на рынке. Если вы недавно закончили университет или курсы программирования и ищете свою первую работу в сфере Java, то вам обязательно нужно купить эту книгу. Она как бы связывает между собой упражнения в программировании и реальную ежедневную работу профессионального инженера-разработчика».

Бен Эванс,
Java-чемпион и главный инженер в New Relic

Рауль-Габриэль Урма, доктор наук, исполнительный директор и основатель Cambridge Spark – ведущей обучающей организации в сфере IT. Автор нескольких книг по программированию, включая такой бестселлер, как «Современный язык Java. Лямбда-выражения, потоки и функциональное программирование».

Ричард Уорбертон, доктор наук, технический директор Opsian – компании, разрабатывающей ультрасовременные инструменты для повышения производительности, сопровождает проект Artio FIX Engine. Также является автором бестселлера и выступает на конференциях.

БОМБОРА
ИЗДАТЕЛЬСТВО

БОМБОРА – лидер на рынке полезных и вдохновляющих книг.
Мы любим книги и создаем их, чтобы вы могли творить, открывать мир, пробовать новое, расти. Быть счастливыми. Быть на волне.

bomborabooks bombora.ru

ISBN 978-5-04-094955-7