Объекты

Стильное ООП

Маттиас Нобак

MANNING

Object Design Style Guide

MATTHIAS NOBACK
FOREWORD BY ROSS TUCK

MANNING
SHELTER ISLAND

Объекты
СТИЛЬНОЕ ООП

Маттиас Нобак

ПИТЕР®

Санкт-Петербург • Москва • Минск

2023

ББК 32.973.23-018
УДК 004.42
Н72

Нобак Маттиас

Н72 Объекты. Стильное ООП. — СПб.: Питер, 2023. — 304 с.: ил. — (Серия «Библиотека программиста»).

ISBN 978-5-4461-1841-0

Хороший объектно-ориентированный код удобно читать, изменять и исправлять. Универсальные практики проектирования объектов, собранные в этой книге, позволят улучшить ваш стиль кодирования. Эти правила подойдут к любому объектно-ориентированному языку, они делают код максимально понятным и надежным, а также повышают производительность как индивидуальных разработчиков, так и команд.

Книга «Объекты. Стильное ООП» познакомит вас с профессиональными техниками написания ОО-кода. Маттиас Нобак раскрывает правила создания объектов, определения методов, изменения и извлечения состояний и многое другое. Все примеры написаны на простом псевдокоде, который легко перевести в любой язык программирования. Кейс за кейсом вы изучите ключевые сценарии и задачи проектирования объектов, а затем шаг за шагом создадите простое веб-приложение, которое покажет, как должны взаимодействовать объекты разных типов.

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

ББК 32.973.23-018
УДК 004.42

ISBN 978-1617296857 англ.
ISBN 978-5-4461-1841-0

Краткое содержание

Оглавление

Моей дочери Джулии:
не позволяй никому говорить, что ты чего-то не можешь из-за того,
что ты девушка

Всем женщинам:
я надеюсь, что вы не побоитесь стать программистами

Всем программистам:
давайте будем приветствовать всех желающих стать одним из нас

Предисловие

Все программисты способны оценить значимость хорошего названия. Название может рассказать, чего стоит и не стоит ожидать. Оно помогает принимать дальнейшие решения.

Но Хорошее Название (с большой буквы) — это нечто большее. Хорошее Название раскрывает душу того, что оно описывает. Оно позволяет представить общую картину, возникшую в голове у создателя, ее суть и смысл. Название Walkie-Talkie отвечает и на вопрос что это, и на вопрос для чего это. Название не обязано быть предельно точным: если муравьи огненные, это не значит, что они действительно пылают. Хорошее Название раскрывает предмет.

Я с удовольствием отмечаю, что у книги, которую вы держите в руках, исключительно Хорошее Название[1].

Вы, возможно, знакомы с руководствами по стилю, которые часто используют журналисты, такими как *AP Stylebook* или *The Chicago Manual of Style*. Подобно им эта книга раскрывает правила и методы, необходимые большим командам, чтобы писать код в понятной и последовательной манере.

Цель Матиаса Нобака, следующая этой модели, понятна и проста. Он не внедряет никаких новых концепций или мудреных инструментов, не рекламирует никаких прорывных подходов. Матиас просто документирует то, чем уже занимается. Он структурирует свой подход к проектированию систем и выделяет из него элементы стиля.

Эти элементы описываются в рамках существующих паттернов, о ряде которых вы наверняка уже слышали. Открытием для меня в этой книге стало то, что эти паттерны редко существуют по отдельности. Примеры, выглядящие в коде рационально, могут не работать в IDE, если не использовать в дополнение другие практики. Например, вспомните, как тяжело проводить модульные тесты без внедрения зависимостей.

---

[1] Оригинальное название книги — «Object Design Style Guide».

Таким образом, «Объекты. Стильное ООП» следует воспринимать как единое целое, а не как сумму его частей. Взятые вместе, описанные паттерны переплетаются и усиливают друг друга. Глубокое погружение в каждую из тем можно найти и в других источниках, а здесь собрана коллекция лучших практик, преобразованная в связный и понятный стиль.

Описание собственного стиля кажется чем-то странным и даже нахальным. В конце концов, почему стоит пользоваться этим стилем, а не вашим или моим? Вспомните, например, стандарты, указывающие, где ставить скобки и пробелы: неважно, какого стиля придерживаться, — главное, чтобы он был единым. Что же касается стиля Матиаса — его достоинство в том, что он уже задокументирован.

Но я убежден, что цель книги — не ограничить пользователя рекомендациями, а послужить отправной точкой. Говорят, что инновации внедряются там, где есть ограничения. Практикуйте этот стиль, улучшайте его, начните с него и продолжите создавать собственную версию, исходя из своей ситуации. В конце концов, хуже газетной статьи, написанной в стиле признания в любви, может быть только признание в любви, написанное в стиле газетной статьи.

Если я вас еще не убедил, эта книга по крайней мере позволит понаблюдать за работой профессионала высокого уровня. Вы встретитесь с подкупающей честностью и уязвимостью. Здесь нет секретного ингредиента или особой техники. Все, что вы видите, — это то, как Матиас делает свою ежедневную работу — не больше, но и не меньше.

В самом деле, в процессе чтения я иногда вспоминал, как читал его код, стоя у него за спиной, и слушал, как он взвешивал концепции и выбирал инструменты. Я указывал на некоторые вещи, которые выбивались из его стиля, а Матиас просто улыбался и говорил: «Ах да, это интересно».

Желаю вам такого же впечатления от прочтения книги, какое получил я.

Росс Так

Вступительное слово

Среди материалов по обучению программированию, с одной стороны, и продвинутым паттернам и принципам проектирования, с другой стороны, не так много посвящено объектно-ориентированным инструментам. Известные рекомендованные книги на эту тему тяжело читать, и зачастую их теорию сложно применить при решении каждодневных задач. К тому же у программистов редко бывает достаточно времени на чтение. В итоге в обучающих материалах зияет брешь.

Расти как программист можно и без чтения книг, по мере получения опыта. Вы научитесь делать правильный выбор. Вы соберете ряд базовых правил, подходящих практически для любой ситуации, что поможет освободить внимание для интересных мест в коде, которые нужно улучшить. Так что все, что невозможно узнать из книг по программированию, на самом деле можно узнать после долгих лет работы с кодом.

Я написал эту книгу, чтобы отчасти устранить эту брешь и предложить читателям нечто, что поможет начать писать более качественный объектно-ориентированный код. Мои предложения довольно компактные и простые. Технически здесь не очень много нового. Но я считаю, что следование этим идеям (или правилам) позволит переключиться с тривиальных на более интересные фрагменты кода, заслуживающие большего внимания. Если все в команде будут работать таким образом, то код во всем проекте станет более единообразным.

Я смело заявляю, что правила проектирования объектов, изложенные в этой книге, улучшат качество объектов и проект в целом. Ее также можно использовать в качестве материала для новичков команды. После того как вы расскажете им о стандартах, применяемых на проекте, и покажете им стили, которыми руководствуется команда, вы можете дать им эту книгу и объяснить, как команда стремится соблюдать грамотный дизайн объектов на всем проекте.

Я желаю вам и всей вашей команде успехов на пути обучения проектированию объектов и, в свою очередь, постараюсь сделать все, чтобы достичь поставленных в этой книге целей.

Благодарности

Прежде чем начать, мне хотелось бы высказать слова благодарности. В первую очередь я благодарю 125 человек, которые приобрели книгу еще до того, как она была закончена. Это вдохновляло! Спасибо также за ваши отзывы, особенно Сергею Лукьяненко, Иосифу Чирилуте (Iosif Chiriluta), Николе Поновицу (Nikola Paunovic), Нико Муку (Niko Mouk), Дэймону Джонсу (Damon Jones) и Мо Халеди (Mo Khaledi). Отдельное спасибо Ремону ван де Кампу (Rémon van de Kamp) за ряд полезнейших комментариев.

Большое спасибо Россу Таку и Лоре Коди (Laura Cody) за тщательный разбор книги. Благодаря вашим предложениям аргументы стали убедительнее, структура — четче, а риски недопонимания — ниже.

Все описанное выше случилось еще до того, как издательство Manning Publications поставило эту книгу в план и начало работать над ее выходом. Майк Стивенс (Mike Stephens), спасибо, что приняли мое предложение. Для меня это был незабываемый опыт. Каждая из консультаций с сотрудниками была весьма полезной: Дедра Хайем (Deirdre Hiam), выпускающий редактор; Энди Кэрол (Andy Carroll), литературный редактор; Кери Хэйлз (Keri Hales), корректор; Александр Драгосавлевич (Aleksandar Dragosavljevic), ревьюер; Таня Вильке (Tanya Wilke), научный редактор; Джастин Колстон (Justin Coulston), корректор. Искренне благодарю всех вас за труд над этим проектом!

Также хотелось бы поблагодарить всех рецензентов: Энджела Р. Родригеса, Бруно Соннино, Карлоса Эзекеля Куротто, Чарльза Сетана, Гаральда Куна, Джозефа Тимоти Лайенса, Джастина Колтона, Марию Джемини, Патрика Рейгана, Пола Гребенка, Саманту Берк, Скотта Штайнмана, Шейна Корнуэла и Стива Уэствуда (Angel R. Rodriguez, Bruno Sonnino, Carlos Ezequiel Curotto, Charles Soetan, Harald Kuhn, Joseph Timothy Lyons, Justin Coulston, Maria Gemini, Patrick Regan, Paul Grebenc, Samantha Berk, Scott Steinman, Shayn Cornwell, Steve Westwood).

Отдельное спасибо Элеше Хайд (Elesha Hyde), ведущему редактору. Она великолепно управляла процессом подготовки проекта, а также внесла значительный вклад в повышение обучающей ценности этой книги.

О книге

ДЛЯ КОГО ЭТА КНИГА

Эта книга предназначена для программистов, владеющих хотя бы базовыми знаниями объектно-ориентированных языков программирования. Вам нужно понимать возможности языка в области классов. Это подразумевает, что вы имеете представление о том, как создать класс, инстанцировать его, расширить, обозначить как abstract, задать метод, вызвать его, задать параметры и их типы, задать возвращаемые типы, задать свойства и их типы и т. д.

Я также рассчитываю, что у вас есть опыт работы со всем этим, пусть даже небольшой. Вы наверняка сможете освоить эту книгу, если прошли базовый курс программирования или давно работаете с классами.

СТРУКТУРА КНИГИ

Написание книги подразумевает переработку больших объемов материала в нечто более сжатое, с чем проще иметь дело. Поэтому креативность без ограничений и правил способна породить хаос. Если добавить правила, то шансы на успешное усвоение материала будут намного выше. Правила помогут легко и не задумываясь принимать локальные решения.

Вот правила, которыми я руководствуюсь в этой книге:

- *Не включать в заголовки глав названия принципов или паттернов*. Идея главы должна быть понятна без попыток запомнить, что весь этот жаргон значит.
- *Короткие разделы*. Я не хочу, чтобы книгу тяжело и нудно читали месяцами. Я хочу, чтобы рекомендации по программированию были готовы к использованию и не были похожи на философские бормотания оракулов. Рекомендация должна быть понятной, и ей легко следовать.
- *Полезное заключение в конце главы*. Если вам нужно быстро перечитать какие-то рекомендации или вы что-то ищете, вам не придется перечитывать всю главу заново. В заключении вы найдете краткое содержание всей главы.

- *Примеры кода сопровождаются советами по его тестированию.* Хорошее проектирование объектов облегчает их тестирование, в то же время объект можно улучшить посредством тестирования. Поэтому имеет смысл оставлять предложения по (модульному) тестированию рядом с рекомендациями по проектированию объектов.

Также обратите внимание на следующие соглашения:

- Словом «клиент» я обозначаю место, откуда класс или метод вызывается. Иногда я называю его «место вызова» или «точка вызова».
- Словом «пользователь» я обозначаю программиста, который использует класс, инстанцируя его и вызывая его методы. Заметьте, что это не пользователь приложения в целом.
- В примерах кода я сокращаю выражения при помощи `// ...` и `/* ... */`. Иногда я использую `//` или `/* ... */`, чтобы указать контекст.

Книга начинается с главы о программировании объектов (глава 1). Эта глава также кратко знакомит с модульным тестированием. В ней объясняется некоторая терминология и рассматриваются концепции объектно-ориентированного программирования.

Само руководство по стилям проектирования начинается со 2-й главы. Сначала мы разграничиваем объекты на сервисы и другие типы. Затем рассматриваем, как создавать объекты-сервисы, а также говорим о том, что после инстанцирования с ними нельзя проводить никакие действия. В главе 3 мы обсудим, как создавать другие объекты, а в главе 4 — какие действия с ними можно осуществлять после инстанцирования.

В главе 5 приводятся некоторые общие рекомендации по написанию методов, которые позволят наделять объекты поведением. Мы рассмотрим два типа действий клиентов над объектами: получение информации из них (глава 6) и запрос выполнения задачи (глава 7). Эти две ситуации применения объектов подразумевают различные правила реализации. В главе 8 объясняется, как разделять модели для записи и для чтения, что поможет разграничивать ответственность за внесение изменений и предоставление информации между множеством объектов.

В главе 9 даются рекомендации для случаев, когда нужно изменить поведение объектов-сервисов. В ней показано, как изменять или улучшать поведение сервисов, объединяя существующие объекты в новые или делая поведение настраиваемым.

Глава 10 — это справочник объектов. В ней показаны различные области (слои) приложения и различные типы объектов, которые встречаются в этих областях.

Книга завершается главой 11, где я составил краткий обзор тем, на которые стоит обратить внимание, если вы захотите узнать больше о проектировании объектов, а также рекомендации для дальнейшего чтения.

И хотя эта книга написана в виде руководства с постепенным изучением всех тем от начала до конца, она может успешно служить и справочником. Поэтому, если вам нужны конкретные рекомендации, смело пропускайте главы, пока не найдете необходимое.

О КОДЕ В КНИГЕ

Примеры кода написаны на обобщенном объектно-ориентированном языке, оптимизированном для чтения широким кругом объектно-ориентированных программистов. Этого языка не существует в реальности, так что код, представленный в книге, нельзя запустить в какой-либо среде исполнения. Но я уверен, что примеры будет легко понять, если у вас есть опыт работы с такими языками программирования, как PHP, Java или C#. Если вы хотите узнать больше об особенностях этого фиктивного языка, загляните в приложение в конце книги.

Некоторые из примеров будут сопровождаться кодом для модульного тестирования. Я опираюсь на тестовый фреймворк xUnit (PHPUnit, JUnit, NUnit и т. п.). Я использую ограниченный набор функций для проверок, обработки исключений или создания тестовых дублеров. Это должно облегчить перенос кода в ваши любимые тестовые фреймворки и библиотеки.

ФОРУМ LIVEBOOK

Приобретая книгу «Объекты. Стильное ООП», вы получаете бесплатный доступ к закрытому веб-форуму издательства Manning (на английском языке), на котором можно оставлять комментарии о книге, задавать технические вопросы и получать помощь от автора и других пользователей. Чтобы получить доступ к форуму, откройте страницу https://livebook.manning.com/book/object-design-style-guide/discussion. Информацию о форумах Manning и правилах поведения на них см. на https://livebook.manning.com/#!/discussion.

В рамках своих обязательств перед читателями издательство Manning предоставляет ресурс для содержательного общения читателей и авторов. Эти обязательства не подразумевают конкретную степень участия автора, которое остается добровольным (и неоплачиваемым). Задавайте автору хорошие вопросы, чтобы он не терял интереса к происходящему! Форум и архивы обсуждений доступны на веб-сайте издательства, пока книга продолжает издаваться.

Об авторе

Маттиас Нобак — профессиональный веб-разработчик (с 2003 года). Он живет в городе Зейст в Нидерландах со своей второй половинкой и детьми: сыном и дочерью.

Маттиас — основатель собственной компании в области веб-разработки, обучения и консультирования под названием Noback's Office. Он вплотную занимается бэкенд-разработкой и программной архитектурой и всегда ищет возможности для улучшения методов проектирования ПО.

С 2011 года ведет блог о программировании на matthiasnoback.nl. Маттиас — автор и других книг: «A Year with Symfony» (Leanpub, 2013), «Microservices for Everyone» (Leanpub, 2017) и «Principles of Package Design»[1] (Apress, 2018). Связаться с Маттиасом можно по электронной почте (info@matthiasnoback.nl) или в Twitter (@matthiasnoback).

[1] Нобак М. «Принципы разработки программных пакетов».

Иллюстрация на обложке

На обложке книги «Объекты. Стильное ООП» изображена женщина с острова Углян рядом с побережьем Хорватии. Иллюстрация взята из французской книги о национальных нарядах «Encyclopedie des Voyages» автора Жака Гассе де сен Савьера (Jacques Grasset de Saint-Sauveur) (1757–1810), опубликованной в 1796 году. Ранее мало кто имел возможность путешествовать ради удовольствия, и такие иллюстрированные руководства пользовались большим спросом и знакомили туристов и исследователей с жителями отдаленных регионов мира, а также с более привычными местными костюмами Франции и Европы.

Разнообразие рисунков в «Encyclopedie des Voyages» наглядно доказывает уникальность и аутентичность разных стран мира и людей, живших около 200 лет назад. Это было время, когда костюмы из двух регионов, разделенных всего несколькими десятками миль, говорили, в каком из этих регионов живет их обладатель, а представителей одного класса, гильдии или племени можно было распознать по тому, какую одежду они носят.

Манера одеваться с тех пор сильно изменилась, и богатое местное разнообразие постепенно ушло в небытие. Сегодня довольно сложно отличить жителей одного континента от другого. Но различия по национальному, этническому и географическому признакам в современном мире все еще существуют и должны признаваться и цениться в обществе. Это то, что стремится культивировать Manning, используя обложки с историческими изображениями, которые привлекают внимание к манере одежды двухсотлетней давности.

От издательства

Ваши замечания, предложения, вопросы отправляйте по адресу comp@piter.com (издательство «Питер», компьютерная редакция).

Мы будем рады узнать ваше мнение!

На веб-сайте издательства www.piter.com вы найдете подробную информацию о наших книгах.

1 Программирование объектов: введение

В этой главе

- ✓ Работа с объектами
- ✓ Модульное тестирование
- ✓ Динамические массивы

Прежде чем приступить к изучению стилей проектирования, давайте обсудим некоторые фундаментальные вопросы программирования с применением объектов. Мы кратко пройдемся по самым важным понятиям и поясним распространенную терминологию, с которой продолжим знакомиться и в последующих главах.

Мы рассмотрим:

- *Классы и объекты* — создание объектов на основе классов, использование конструктора, статические и объектные методы, статические методы фабрик для создания новых экземпляров и выдача исключений в конструкторе (раздел 1.1).
- *Состояние* — объявление свойств private и public, задание им значений, константы, а также изменяемые и неизменяемые состояния (раздел 1.2).
- *Поведение* — private- и public-методы, передача значений в качестве аргументов и исключения `NullPointerException` (раздел 1.3).

- *Зависимости* — их инициализация, обнаружение и передача в конструктор в качестве аргумента (раздел 1.4).
- *Наследование* — интерфейсы, абстрактные классы, переопределение и финальные классы (раздел 1.5).
- *Полиморфизм* — один интерфейс, разное поведение (раздел 1.6).
- *Композиция* — присвоение объектов свойствам и создание более сложных объектов (раздел 1.7).
- *Организация классов* (раздел 1.8).
- *Оператор возврата и исключения* — возвращение значения из метода, исключения в методе, обработка исключений, объявление собственных классов исключений (раздел 1.9).
- *Модульное тестирование* — паттерн Arrange-Act-Assert, тестирование на сбои и использование дублеров тестов для замещения зависимостей (раздел 1.10).
- *Динамические массивы* — их использование для создания списков или маппингов (раздел 1.11).

Если вы знакомы с этими темами, можете смело пропускать эту главу и переходить к главе 2. Если только некоторые темы представляют для вас интерес, просмотрите соответствующие разделы. Если же вы начинающий объектно-ориентированный программист, я рекомендую прочесть главу целиком.

1.1. КЛАССЫ И ОБЪЕКТЫ

Поведение объекта в среде исполнения задается объявлением класса. На основе класса можно создавать любое количество объектов. В следующем листинге показан простой класс, у которого нет состояния или поведения, но который можно инстанцировать.

Листинг 1.1. Минимально жизнеспособный класс

```
class Foo
{
    // Здесь ничего нет
}

object1 = new Foo();
object2 = new Foo();

object1 == object2 // false
```

Два экземпляра одного класса нельзя считать равными

После того как вы создадите экземпляр, можно будет вызывать его методы.

Листинг 1.2. Вызов метода экземпляра

```
class Foo
{
    public function someMethod(): void
    {
        // Что-нибудь сделать
    }
}

object1 = new Foo();
object1.someMethod();
```

Обычный метод, такой как `someMethod()`, можно вызвать только на *экземпляре* класса. Такой метод называется *объектным методом*. Можно также задать методы, вызываемые без наличия экземпляра. Они называются *статическими методами*.

Листинг 1.3. Объявление статического метода

```
class Foo
{
    public function anObjectMethod(): void
    {
        // ...
    }

    public static function aStaticMethod(): void
    {
        // ...
    }
}

object1 = new Foo();
object1.anObjectMethod();

Foo.aStaticMethod();
```

anObjectMethod() можно вызвать только на экземпляре SomeClass

aStaticMethod() можно вызвать без экземпляра

Кроме объектных и статических методов, класс может иметь специальный метод — *конструктор*. Этот метод вызывается до того, как объекту присваивается ссылка. Если необходимо подготовить объект к использованию, это можно сделать в конструкторе.

Листинг 1.4. Объявление метода-конструктора

```
class Foo
{
    public function __construct()
    {
        // Подготовить объект
    }
}

object1 = new Foo();
```

__construct будет явно вызван перед тем, как object1 будет присвоен экземпляр Foo

Можно помешать созданию объекта, вызвав *исключения* в конструкторе, как показано в листинге ниже. Подробнее об исключениях см. в разделе 1.9.

Листинг 1.5. Обработка исключения в конструкторе

```
class Foo
{
    public function __construct()
    {
        throw new RuntimeException();
    }
}

try {
    object1 = new Foo();
} catch (RuntimeException exception) {
    // `object1` будет неопределен
}
```

Инстанцировать Foo невозможно, так как его конструктор всегда будет выдавать исключение

Стандартный способ инстанцирования объекта класса — использование оператора `new`, как вы уже, наверно, заметили. Также можно задать статичный *фабричный метод* класса, который будет возвращать новый экземпляр этого класса.

Листинг 1.6. Объявление статичного фабричного метода

```
class Foo
{
    public static function create(): Foo
    {
        return new Foo();
    }
}

object1 = Foo.create();
object2 = Foo.create();
```

Метод `create()` нужно объявлять статичным, так как он вызывается на классе, а не на экземпляре класса.

1.2. СОСТОЯНИЕ

Объект может хранить данные. Эти данные могут содержаться в *свойствах*. Свойства будут иметь *имя* и *тип*; им можно присвоить значения в любой момент после создания объекта. Обычно свойства задаются в конструкторе.

Листинг 1.7. Объявление свойств и присвоение им значений

```
class Foo
{
    private int someNumber;
```

```
    private string someString;

    public function __construct()
    {
        this.someNumber = 10;
        this.someString = 'Hello, world!';
    }
}

object1 = Foo.create();
```

После инициализации свойствам someNumber и someString будут присвоены значения 10 и Hello, world! соответственно

Данные, содержащиеся в объектах, также называют *состоянием* объекта. Если эти данные необходимо жестко закрепить, как в предыдущем листинге, лучше присвоить значение изначально или объявить для него константу.

Листинг 1.8. Объявление констант

```
class Foo
{
    private const int someNumber = 10;
    private someString = 'Hello, world!';
}
```

В языке программирования может быть другой синтаксис. Например, в Java вы увидите final private int someNumber

С другой стороны, если значение свойства должно быть *переменным*, можно позволить клиенту передавать значение для аргумента конструктора. Добавляя параметр в конструктор, вы вынуждаете клиентов предоставлять значение при создании экземпляра класса.

Листинг 1.9. Добавление аргумента в конструктор

```
class Foo
{
    private int someNumber;

    public function __construct(int initialNumber)
    {
        this.someNumber = initialNumber;
    }
}

object1 = new Foo(); // не работает
object2 = new Foo(20);
```

Невозможно создать экземпляр Foo без предоставления значения для аргумента initialNumber

Должно сработать, здесь новому экземпляру класса Foo присваивается начальное значение 20

Если присвоить свойствам `someNumber` и `someString` модификатор `private`, они станут доступными только внутри экземпляров класса `Foo`. Это называется *инкапсуляцией*. Существуют и другие модификаторы для свойств: `protected` (см. раздел 1.5) и `public`. Присваивая свойству модификатор `public`, вы предоставляете любому клиенту доступ к нему.

Листинг 1.10. Объявление свойства с модификатором public

```
class Foo
{
    public const int someNumber;

    public string someString;

    // ...
}

object1 = new Foo();
number = object1.someNumber;
object2.someString = 'Cliché';
```

Так как свойство someNumber задано как константа, его значение менять нельзя, но хотя бы можно получить

someString не константа, но это свойство задано с модификатором public, поэтому здесь можно менять его значение

МОДИФИКАТОРОМ ПО УМОЛЧАНИЮ ДЛЯ СВОЙСТВ ДОЛЖЕН БЫТЬ PRIVATE

В общем случае предпочтительно использовать модификатор `private`. Ограничение доступа к данным объекта помогает скрывать детали его реализации. Так можно гарантировать, что клиентам не придется полагаться на специфичные данные объекта и они будут общаться с объектом через объявленные публичные методы (подробнее о методах см. в разделе 1.3). Мы разберем эту тему подробнее в следующих главах, например в разделах 6.3 и 9.8.

Инкапсуляция свойств (а также методов, см. раздел 1.3) основана на классах. Это означает, что если свойство имеет модификатор `private`, то к этому свойству имеют доступ все экземпляры класса, включая тот экземпляр, которому оно присвоено.

Листинг 1.11. Обращение к свойству private другого экземпляра

```
class Foo
{
    private int someNumber;

    // ...

    public function getSomeNumber(): int
    {
        return this.someNumber;
    }

    public function getSomeNumberFrom(Foo other): int
    {
        return other.someNumber;
    }
}

object1 = new Foo();
object2 = new Foo();

object2.getSomeNumberFrom(object1);
```

Foo, конечно же, имеет доступ к собственному свойству someNumber

Foo также имеет доступ к свойству someNumber других экземпляров Foo

Здесь вернется значение свойства someNumber

Объект называется *изменяемым*, когда значение свойства объекта может меняться в течение всего времени существования объекта. Если свойства объекта не могут меняться после инициализации, то объект считается *неизменяемым*. Листинг ниже иллюстрирует оба случая.

Листинг 1.12. Изменяемые и неизменяемые объекты

```
class Mutable
{
    private int someNumber;

    public function __construct(int initialNumber)
    {
        this.someNumber = initialNumber;
    }

    public function increase(): void
    {
        this.someNumber = this.someNumber + 1;
    }
}

class Immutable
{
    private int someNumber;

    public function __construct(int initialNumber)
    {
        this.someNumber = initialNumber;
    }

    public function increase(): Immutable
    {
        return new Immutable(someNumber + 1);
    }
}

object1 = new Mutable(10);
object1.increase();

object2 = new Immutable(10);
object2 = object2.increase();
```

Вызов increase() у Mutable изменит состояние объекта, модифицируя значение свойства someNumber

Вызов increase() у immutable не изменит состояния object2. Вместо этого мы получим новый экземпляр с увеличенным значением someNumber

В разделе 4.4 мы подробнее рассмотрим изменяемые объекты и узнаем, как делать их неизменяемыми.

1.3. ПОВЕДЕНИЕ

Кроме состояния объект обладает поведением, которым клиенты могут пользоваться. Это поведение объекта задается методами класса. Методы с модификатором

`public` доступны клиентам объекта. Их можно вызывать в любое время после создания самого объекта.

Некоторые методы возвращают сущности тому, кто его вызывает. В таком случае в качестве *возвращаемого типа* будет объявлен тип сущности. Некоторые методы ничего не возвращают. В таком случае возвращаемым типом будет `void`.

Листинг 1.13. Поведение объекта задается публичными методами

```
class Foo
{
    public function someMethod(): int
    {
        return /* ... */;
    }

    public function someOtherMethod(): void
    {
        // ...
    }
}

object1 = new Foo();
value = object1.someMethod();

object1.someOtherMethod();
```

someMethod() возвращает целое число, которое можно присвоить переменной

someOtherMethod() не возвращает ничего конкретного, поэтому клиент не сможет ничего присвоить переменной

Класс также может содержать объявления методов с модификатором `private`. Это работает так же, как и для приватных переменных. Любой экземпляр данного класса может вызывать приватные методы на любых других экземплярах одного и того же класса, включая себя самого. Тем не менее приватные методы часто используются для выполнения мелких шагов в большом процессе.

Листинг 1.14. Приватные методы

```
class Foo
{
    public function someMethod(): int
    {
        value = this.stepOne();

        return this.stepTwo(value);
    }

    private function stepOne(): int
    {
        // ...
    }

    private function stepTwo(int value): int
    {
        // ...
    }
}
```

ПРОВЕРЯЙТЕ НАЛИЧИЕ АРГУМЕНТОВ СО ЗНАЧЕНИЕМ NULL

В некоторых языках клиенты могут передавать значение `null` в качестве аргумента, даже если тип параметра был явно задан. Так что в примере листинга 1.15 аргумент для `bar` может оказаться `null`, даже если его тип явно указан — `Bar`. Попытка вызывать метод `doSomething()` на объекте `bar` приведет к выдаче исключения `NullPoinerException`. Поэтому всегда нужно проверять наличие аргументов со значением `null` либо обращать внимание на предупреждения компилятора или статического анализатора о потенциальных исключениях `NullPointerException`.

Вымышленный язык программирования, используемый в этой книге, *не позволяет* передавать `null` в качестве аргумента. Если мы хотим разрешить передачу `null`, это необходимо задать явно при помощи вопросительного знака (`?`) после объявления параметра метода. Таким же образом это работает и для типов свойств и возвращаемых типов:

```
class Foo
{
    private string? foo;

    private function someOtherMethod(Bar? bar): Baz?
    {
        // ...
    }
}
```

Так же как параметры конструкторов, можно задавать параметры методам. Клиенту придется предоставить специфичное значение в качестве аргумента при вызове метода. Сам метод может использовать значение для дальнейших действий: передать его вспомогательным объектам или использовать его для изменения значения свойства.

Листинг 1.15. Несколько способов применения аргументов метода

```
class Foo
{
    private int number;

    public function setNumber(int newNumber): void
    {
        this.number = newNumber;
    }

    private function multiply(int other): int
    {
        return this.number * other;
    }
```

Здесь аргумент newNumber станет новым значением свойства number

В этом случае аргумент other будет умножаться на текущее значение свойства number

```
    private function someOtherMethod(Bar bar): void
    {
        bar.doSomething();
    }
}
```

Здесь в качестве аргумента передается другой объект, и Foo даже может вызвать его метод

1.4. ЗАВИСИМОСТИ

Если объекту `Foo` нужен объект `Bar` для выполнения задачи, то `Bar` называют зависимостью `Foo`. Существуют три способа проверить, что у `Foo` есть доступ к зависимости `Bar`.

- `Foo` может самостоятельно инициализировать `Bar`.
- `Foo` может получить экземпляр `Bar` из известного источника.
- `Foo` может получить экземпляр `Bar` во время вызова конструктора.

В листинге ниже проиллюстрирован каждый из этих трех вариантов.

Листинг 1.16. Различные способы получения доступа Foo к экземпляру Logger

```
class Foo
{
    public function someMethod(): void
    {
        logger = new Logger();
        logger.debug('...');
    }
}
```

Foo инстанцирует Logger при необходимости

```
class Foo
{
    public function someMethod(): void
    {
        logger = ServiceLocator.getLogger();
        logger.debug('...');
    }
}
```

Foo получает Logger из известного источника

```
class Foo
{
    private Logger logger;

    public function __construct(Logger logger)
    {
        this.logger = logger;
    }

    public function someMethod(): void
    {
        this.logger.debug('...');
    }
}
```

Foo получает экземпляр Logger, предоставленный в качестве аргумента конструктора

Как обращаться с зависимостями, мы рассмотрим подробнее в разделе 2.2. Сейчас же достаточно знать, что получение зависимостей из известного источника называется *локализацией сервисов*, а получение зависимостей в качестве аргументов конструктора — *внедрением зависимостей*.

1.5. НАСЛЕДОВАНИЕ

Можно также задавать только часть класса и позволить другим классам его расширять. Например, у вас может быть класс без свойств и методов, только с сигнатурами методов. Такие классы обычно называют *интерфейсами*, и многие ООП-языки позволяют их таковыми объявлять. Класс затем использует интерфейс и предоставляет фактические реализации методов, определенных интерфейсом.

Листинг 1.17. Реализации Bar и Baz интерфейса Foo

```
interface Foo
{
    public function foo(): void;
}

class Bar implements Foo
{
}

class Baz implements Foo
{
    public foo(): void
    {
        // ...
    }
}
```

Интерфейс Foo объявляет метод foo(), но не предоставляет его реализацию

Класс Bar неверно реализует интерфейс Foo, так как в нем нет реализации метода foo()

Класс Baz правильно реализует интерфейс Foo, так как в нем представлена реализация метода foo()

В интерфейсах нельзя задать конкретную реализацию, но она может задаваться в *абстрактных классах*. Они позволяют предоставлять реализацию методов и сигнатуры методов. Абстрактный класс нельзя инстанцировать, но его можно расширить в другом классе, в котором будут реализовываться абстрактные методы.

Листинг 1.18. Baz расширяет абстрактный класс Foo

```
abstract class Foo
{
    abstract public function foo(): void;

    public function bar(): void
    {
        // ...
    }
}
```

Метод foo() абстрактный и должен быть реализован расширяющим классом

Класс Foo задает реализацию методу bar()

```
class Baz extends Foo
{
    public function foo(): void
    {
    }
}
```

Baz — корректная реализация класса Foo, так как в нем представлена реализация ранее объявленного метода foo()

И наконец, в классе могут быть заданы реализации всех его методов, а другие классы могут их расширять и переопределять.

Листинг 1.19. Класс Bar расширяет класс Foo и частично меняет его поведение

```
class Foo
{
    public function bar(): void
    {
        // сделать что-то
    }
}

class Bar extends Foo
{
    public function bar(): void
    {
        // сделать что-то еще
    }
}
```

Foo — обычный класс, без абстрактных методов

Bar расширяет Foo, который теперь является родительским классом. Bar может менять поведение его метода bar()

Foo — обычный класс, без абстрактных методов

Классы, которые расширяют другие классы, получают доступ к родительским методам с модификаторами доступа `public` и `protected`.

Листинг 1.20. Доступ к методам с модификаторами доступа public и protected

```
class Foo
{
    public function foo(): void
    {
        // сделать что-то
    }

    protected function bar(): void
    {
    }

    private function baz(): void
    {
    }
}

class Bar extends Foo
{
    public function someMethod(): void
```

```
    {
        $this->foo();   <-- foo() доступен, так как это метод с модификатором public
        $this->bar();   <-- bar() доступен, так как это метод с модификатором protected
        //$this->baz();   <-- baz() недоступен, так как это приватный метод
    }
}
```

Подклассы могут также переопределять родительские методы с модификаторами `public` и `protected`.

Листинг 1.21. Переопределение методов с модификаторами public и protected

```
class Foo
{
    public function foo(): void
    {
        // сделать что-то
    }

    protected function bar(): void
    {
    }

    private function baz(): void
    {
    }
}

class Bar extends Foo
{
    public function foo(): void   <-- Метод foo() можно переопределить, так как это метод с модификатором public
    {
        // ...
    }

    protected function bar(): void   <-- Метод bar() можно переопределить, так как это метод с модификатором protected
    {
        // ...
    }

    private function baz(): void   <-- Метод baz() нельзя переопределять, так как это приватный метод
    {
        // не работает
    }
}
```

В этой книге наследованию уделено не очень много внимания, несмотря на то что это крайне важный аспект ООП. На практике наследование зачастую приводит к усложнению дизайна. В книге мы в основном будем использовать наследование в двух случаях:

- при объявлении интерфейсов для зависимостей;
- при объявлении иерархии объектов, как, например, при объявлении кастомных исключений, которые расширяют встроенные классы исключений.

В большинстве случаев старайтесь не допускать расширений из классов. Для этого используйте ключевое слово `final` перед названием класса. Подробнее см. в разделе 9.7.

Листинг 1.22. Класс Bar нельзя расширить

```
final class Bar
{
    // ...
}

class Baz extends Bar // не работает
{
    // ...
}
```

Bar обозначен ключевым словом final, поэтому его запрещено расширять

1.6. ПОЛИМОРФИЗМ

Полиморфизм — это одна из основополагающих концепций ООП. Полиморфизм означает, что если для параметра задан определенный класс в качестве типа, то любой объект, являющийся экземпляром этого класса, можно передавать в качестве допустимого аргумента. Например, в следующем листинге любой экземпляр `Foo` можно передать в качестве аргумента методу `bar()`:

Листинг 1.23. Любой экземпляр Foo будет допустим для метода bar()

```
class Foo
{
    // ...
}

final class Bar
{
    public function bar(Foo foo): void
    {
        foo.someMethod();
    }
}
```

Так как экземпляры Foo могут иметь различную конфигурацию или внутреннее состояние, то теоретически каждый из них может вести себя по-своему. Это значит, что можно влиять на поведение `bar()`, не изменяя сам метод `bar()`.

Еще больше влияния на поведение классов можно оказывать при помощи подклассов. Мы уже рассмотрели наследование и то, как использовать его для изменения поведения, унаследованного родительским классом, при помощи (частичного) переопределения в расширяющем классе. Любой объект, являющийся экземпляром расширения класса `Foo`, также будет считаться экземпляром класса `Foo`. Это также автоматически делает любой экземпляр расширения класса `Foo` допустимым аргументом для параметров типа `Foo`.

Как вы узнаете в главе 9, применять подклассы для изменения поведения объектов обычно не рекомендуется. В большинстве ситуаций лучше использовать полиморфизм в типе параметра интерфейса. В обоих вариантах код будет выглядеть одинаково (см. листинг 1.24), но во втором случае `Foo` представляет собой интерфейс.

Листинг 1.24. Любой экземпляр Foo будет приемлем для метода bar()

```
interface Foo  <-- Теперь Foo — интерфейс
{
    // ...
}

final class Bar
{
    public function bar(Foo foo): void
    {
        foo.someMethod();
    }
}
```

1.7. КОМПОЗИЦИЯ

Кроме демонстрации полиморфизма в листинге 1.25 показано, как использовать экземпляр `Foo` для работы в другом объекте (`Bar`). Если `Foo` — это сервис, то его можно предоставлять объекту класса `Bar` в качестве аргумента конструктора. `Bar` затем может назначить объект класса `Foo` своим свойством.

Листинг 1.25. Предоставленный экземпляр Foo передается в качестве свойства объекта Bar

```
final class Bar
{
    private Foo foo;

    public function __construct(Foo foo)
    {
        this.foo = foo;
    }
}
```

Передача объекта другому объекту в качестве свойства называется *композицией объектов*. При этом возникает более сложный объект, состоящий из множества простых. Композицию можно использовать вместе с полиморфизмом для создания объектов, содержащих другие объекты с известными типами (интерфейсов) без указания конкретных классов.

Композицию можно использовать для объектов-сервисов, внедряя возможность частичного конфигурирования их поведения, а также применять вместе с другими типами объектов, например с сущностями (их иногда называют *моделями*), для работы со связанными внутренними элементами. Например, в объекте `Order`, который содержит объекты `Line`, композиция позволяет устанавливать связь между заказами и элементами заказа. В этом случае клиенту может понадобиться предоставить не один объект `Line`, а целый ряд (массив) объектов `Line`.

Листинг 1.26. Свойству объекта Order присваивается множество объектов Line

```
final class Order
{
    private array lines;

    public function __construct(array lines)
    {
        this.lines = lines;  <-- Каждый элемент в lines — это объект Line
    }
}
```

1.8. ОРГАНИЗАЦИЯ КЛАССОВ

В языках программирования обычно существует несколько возможностей для организации классов: директории, пространства имен, компоненты, модули, пакеты и т. п. В некоторых языках даже допускается хранить приватные классы в модулях или пакетах. Так же как область видимости переменных и методов, это помогает ограничить степень связанности модулей. Эта книга не содержит конкретных правил организации классов в большие группы — она описывает правила проектирования самих классов. Если вам интересны принципы организации классов на уровне компонентов, советую прочесть другую мою книгу, *Principles of Package Design*[1] (Apress, 2018).

1.9. ОПЕРАТОР ВОЗВРАТА И ИСКЛЮЧЕНИЯ

Когда вы вызываете методы, они обычно выполняются последовательно сверху вниз, пока не достигнут оператора возврата `return` или конца метода. Если вы

[1] Нобак М. «Принципы разработки программных пакетов: Проектирование повторно используемых компонентов».

хотите прервать выполнение метода, поместите в месте прерывания оператор `return`, который гарантирует, что оставшиеся выражения метода не будут выполняться.

Листинг 1.27. Оператор return предотвращает дальнейшее выполнение метода

```
final class Foo
{
    public function someMethod(): void
    {
        if (/* прервать здесь? */) {
            return;
        }

        // ...
    }

    public function someOtherMethod(): bool
    {
        if (/* прервать здесь? */) {
            return false;
        }

        // ...

        return true;
    }
}
```

Метод с возвращаемым типом void не возвращает ничего

Метод с указанным возвращаемым типом возвращает конкретные значения

Еще один способ прервать выполнение метода — *вызов исключения*. Исключение — это особый тип объекта, который при инстанцировании собирает информацию о том, где он был инстанцирован и что этому предшествовало (так называемое *отслеживание стека*). Обычно исключение свидетельствует о сбое в программе, например:

- о предоставлении неверных аргументов;
- отсутствии значения в массиве для данного ключа;
- недоступности некоторых внешних сервисов.

В листинге ниже показано, как вызвать исключение.

Листинг 1.28. Выдача исключения также предотвращает дальнейшее выполнение метода

```
final class Foo
{
    public function someMethod(): void
    {
        if (/* прервать здесь? */) {
```

```
            throw new RuntimeException(
                'Что-то пошло не так'   <-- Можно прописать сообщение для исключения
            );
        }

        // ...
    }
}
```

Как только становится понятно, что метод не сможет выполнять задачу корректно, в нем нужно объявить выдачу исключения. В отличие от простого оператора `return`, при выдаче исключения метод не возвращает значений. Выполнение прекращается и может быть продолжено только клиентом, у которого вызов метода был объявлен внутри блока `try/catch`. Ниже в листинге показано, как это работает.

Листинг 1.29. Клиент может восстановить выполнение метода после выдачи исключения, если будет использовать блок try/catch

```
foo = new Foo();

try {
    foo.someMethod();
} catch (Exception) {
    // ...
}
```

Если someMethod() выдаст исключение, catch() его поймает и выполнение остальных операций продолжится

В разных языках программирования предусматриваются свои классы исключений. Они образуют некоторую иерархию — например, `RuntimeException` расширяет `Exception`, `InvalidArgumentException` расширяет `LogicException`. Также можно объявить собственные классы исключений. Но они всегда должны расширять один из встроенных классов исключений. Пример — в листинге 1.30.

Листинг 1.30. Объявление собственного исключения

```
final class CanNotFindFoo extends RuntimeException
{
    // ... 1((CO25-1))
}

final class Foo
{
    public function someMethod(): void
    {
        if (/* прервать здесь?  */) {
            throw new CanNotFindFoo();
        }

        // ...
    }
}
```

Исключения — важный аспект проектирования объектов. Они входят в набор типов поведения, которое клиент может ожидать от объекта. Подробнее об исключениях мы поговорим позже, в разделе 5.2.

1.10. МОДУЛЬНОЕ ТЕСТИРОВАНИЕ

Недостаточно просто объявить объекты посредством написания классов. Объекты служат определенной цели: они будут использоваться для выполнения конкретной задачи или для предоставления ответа на конкретный вопрос. Для надежности объект нужно наделить поведением, которого ожидает клиент. Бесспорно, можно написать код, скомпилировать его и запустить приложение, а затем узнать, получен ли ожидаемый результат. Но более грамотный подход — написать скрипт, который инстанцирует объект, вызывает его метод и затем сравнивает результат с ожидаемым.

Фреймворки модульного тестирования поддерживают такой подход со скриптами. Фреймворк будет искать классы специфичного типа, называемые *тестовыми* классами. Затем он инстанцирует каждый класс и вызывает каждый метод, который обозначен как тестовый (методы с аннотацией `@test`).

Базовая структура каждого метода выстроена по паттерну Arrange-Act-Assert:

- **Arrange** (подготовка) — приведение тестируемого объекта (также называемого SUT, subject under test) в определенное известное состояние;
- **Act** (действие) — вызов одного из его методов;
- **Assert** (проверка) — оценка конечного состояния.

В листинге 1.31 показан простой класс с сопутствующими модульными тестами.

Листинг 1.31. Простой класс с несколькими модульными тестами

```
final class Foo
{
    private int someNumber;

    public function __construct(int startWith)
    {
        this.someNumber = startWith;
    }

    public function increment(): void
    {
        this.someNumber++;
    }

    public function someNumber(): int
```

```
    {
        return this.someNumber;
    }
}

final class FooTest
{
    /**
     * @test
     */
    public function you_can_start_with_a_given_number(): void
    {
        // Arrange
        foo = new Foo(10);

        // Act
        // Assert
        assertEquals(10, foo.someNumber());
    }

    /**
     * @test
     */
    public function you_can_increment_the_number(): void
    {
        // Arrange
        foo = new Foo(10);

        // Act
        foo.increment();

        // Assert
        assertEquals(11, foo.someNumber());
    }
}
```

Здесь не выполняется действие, это проверка состояния объекта

Вызываем метод increment(), что является действием. После этого проверяем, находится ли объект в ожидаемом состоянии

Если во втором тесте значение, возвращаемое методом `someNumber()`, а именно 11, равно ожидаемому, то все в порядке. Поток выполнения будет продолжаться под контролем тестового фреймворка. Если же `someNumber()` будет не полностью реализован или реализован с ошибками, то результатом вызова метода `assertEquals()` станет исключение. Если, например, `someNumber()` вернет значение 20, то тестовый фреймворк зафиксирует этот тест как *проваленный*. Когда вы исправите проблему и снова запустите тест, то он будет *пройден*.

`assertEquals()` и похожие проверки, такие как `assertTrue()`, `assertNull()` и т. п., обычно уже содержатся в тестовом фреймворке. Их можно использовать для оценки возвращаемых значений вызванных методов. Еще может пригодиться проверка случаев, когда метод должен выдавать контролируемую *ошибку*. Например, если вы хотите наложить ограничение на число, предоставляемое конструктору `Foo` в качестве аргумента (например, оно должно быть больше или равно 0), то следует

проверить, что предоставление Foo отрицательного числа приведет к выдаче исключения. В листинге 1.32 показано, как это сделать.

Листинг 1.32. Тестирование на ошибки

```
final class Foo
{
    private int someNumber;

    public function __construct(int startWith)
    {
        if (startWith < 0) {
            throw new InvalidArgumentException(
                'Первое число не должно быть отрицательным'
            );
        }
        this.someNumber = startWith;
    }

    // ...
}

final class FooTest
{
    /**
     * @test
     */
    public function you_cannot_start_with_a_negative_number(): void
    {
        try {
            new Foo(-10);
            throw new RuntimeException(
                'Конструктор должен завершиться ошибкой'
            );
        } catch (Exception exception) {
            if (exception.className != InvalidArgumentException.className) {
                throw new RuntimeException(
                    'Ожидается другой тип исключения'
                );
            }

            assertContains('negative', exception.getMessage());
        }
    }

    // ...
}
```

Если в результате инстанцирования Foo с отрицательным числом не выдается исключение, необходимо считать тест проваленным

Если выданный класс исключения не совпадает с ожидаемым, необходимо считать тест проваленным

Наконец, проверяем, что сообщение исключения содержит ожидаемое ключевое слово

Это стандартный код на каждый сценарий тестирования на ошибки. К счастью, в тестовых фреймворках обычно уже имеются инструменты для тестирования исключений — что-то вроде сервисной функции `expectException()`, представленной в следующем листинге.

Листинг 1.33. Сервисная функция для тестирования на ошибки

```
final class FooTest
{
    /**
     * @test
     */
    public function you_cannot_start_with_a_negative_number(): void
    {
        expectException(
            InvalidArgumentException.className,
            'negative',
            function () {
                new Foo(-10);
            }
        );
    }

    // ...
}
```

Ожидаемое ключевое слово в сообщении

Ожидаемый класс исключения

Анонимная функция, вызывающая метод, в котором ожидается ошибка

Если тестируемый объект имеет зависимости, то, возможно, вы не захотите использовать настоящие зависимости. Например, эти зависимости могут начать вносить изменения в базу данных или отправлять электронные письма. Каждый запуск тестов будет чреват этими нежелательными последствиями. В такой ситуации нужно заменить реальные зависимости объектами, похожими на настоящие зависимости, но с измененной частью поведения. Ниже представлен пример.

Листинг 1.34. Использование тестового дублера

```
interface Mailer 1((CO28-1))
{
    public function sendWelcomeEmail(UserId userId): void;
}

final class ActualMailer implements Mailer
{
    public function sendWelcomeEmail(UserId userId): void
    {
        // Отправить электронное письмо
    }
}

final class StandInMailer implements Mailer
{
    public function sendWelcomeEmail(UserId userId): void
    {
        // Ничего не делать
    }
}
```

Задайте интерфейс для зависимости и предоставьте его замену

```
class Foo
{
    private Mailer mailer;

    public function __construct(Mailer mailer)
    {
        this.mailer = mailer;
    }
}

// В тесте:
foo = new Foo(new StandInMailer());
```

В тесте можно инстанцировать Foo, предоставляя эту замену в качестве аргумента конструктора

Если вам захочется проверить, что `Foo` действительно вызывает метод `sendWelcomeEmail()`, можете использовать особый вид замены под названием *имитация* (mock). Тестовые фреймворки обычно содержат специальные инструменты для настройки таких имитаций и проведения нужных проверок. В листинге ниже показан пример применения имитации без использования специальных инструментов.

Листинг 1.35. Использование простой имитации для проверки успешного вызова метода

```
final class MockMailer implements Mailer
{
    private bool hasBeenCalled = false;

    public function sendWelcomeEmail(UserId userId): void
    {
        this.hasBeenCalled = true;
    }

    public function hasBeenCalled(): bool
    {
        return this.hasBeenCalled;
    }
}

class Foo
{
    private Mailer mailer;

    public function __construct(Mailer mailer)
    {
        this.mailer = mailer;
    }

    public function someMethod(): void
    {
        this.mailer.sendWelcomeEmail();
    }
```

Единственное, что делает эта имитация, — фиксирует, что метод sendWelcomeEmail() действительно вызывается

```
}

// В тесте:
mockMailer = new MockMailer();
foo = new Foo(mockMailer);

foo.someMethod();

assertTrue(mockMailer.hasBeenCalled());
```

Мы предоставляем имитацию в качестве зависимости

В конце теста проверим, что имитация получила запрос на вызов метода sendWelcomeEmail()

Тестовые дублеры мы рассмотрим подробнее в разделах 6.6 и 7.7.

О тестировании и подходах к нему можно рассказать еще очень многое, но это не входит в задачи книги[1]. В этом разделе я хотел показать некоторые базовые техники, используемые в модульном тестировании. Далее будут представлены более подробные примеры и выводы по этой теме.

1.11. ДИНАМИЧЕСКИЕ МАССИВЫ

Эта книга служит руководством по стилям проектирования объектов. А значит, большинство примеров кода будут раскрывать особенности классов, свойств и методов. Код в самих методах будет не настолько важен, поэтому я постарался, чтобы он был как можно проще. Тем не менее для некоторых примеров мне нужны были структуры данных, такие как ассоциативные массивы и списки. Если использовать соответствующие классы `List` и `Map` и явное объявление типов ключей и значений элементов, то код станет слишком громоздким, поэтому я решил вместо этого использовать сущность под названием *динамический массив*.

Динамический массив — это структура данных, которую можно использовать для создания и списков, и ассоциативных массивов. *Список* — это коллекция значений, составленных в определенном порядке. Список можно перебирать и получать из него значения по указателю — целому числу от нуля и до конца списка.

Листинг 1.36. Использование динамического массива в качестве списка

```
emptyList = [];

listOfStrings = ['foo', 'bar'];
```

[1] Другие хорошие книги по этой теме: Кент Бек (Kent Beck) «Test-Driven Development: By Example» («Экстремальное программирование: разработка через тестирование»); Стив Фриман (Steve Freeman), Нэт Прайс (Nat Pryce) «Growing Object-Oriented Software, Guided by Tests» (не издавалась на русском языке); а также Джерард Месарош (Gerard Meszaros) «XUnit Test Patterns» («Шаблоны тестирования xUnit. Рефакторинг кода тестов»).

```
// Цикл по списку:
foreach (listOfStrings as key => value) {
    // Первый: key = 0, value = 'foo'
    // Второй: key = 1, value = 'bar'
}

// Если ключ не подходит:
foreach (listOfStrings as value) {
    // Первый: value = 'foo'
    // Второй: value = 'bar'
}

// Получение значения по определенному индексу:
fooString = listOfStrings[0];
barString = listOfStrings[1];

// Добавление элементов в список:
listOfStrings[] = 'baz';
```

Ассоциативный массив — это коллекция значений, но значения в ней не упорядочены. Вместо этого каждое значение добавляется в массив вместе с уникальным ключом типа `string`. При помощи ключа можно получать значения массива. Ниже в листинге показано, как использовать динамический массив в качестве ассоциативного массива.

Листинг 1.37. Использование динамического массива в качестве ассоциативного

```
emptyMap = [];

mapOfStrings = [
    'foo' => 'bar',
    'bar' => 'baz'
];

// Цикл по массиву:
foreach (mapOfStrings as key => value) {
    // Первый: key = 'foo', value = 'bar'
    // Второй: key = 'bar', value = 'baz'
}

// Получение значения по определенному индексу:
fooString = mapOfStrings['foo'];
barString = mapOfStrings['bar'];

// Добавление элементов в массив:
mapOfStrings['baz'] = 'foo';
```

Эти массивы называются *динамическими*, чтобы не приходилось объявлять тип для ключей и их значений и не задавать их начальный размер. Динамические массивы будут автоматически увеличиваться при добавлении в них значений.

ЗАКЛЮЧЕНИЕ

- Объекты можно инстанцировать на основе класса.
- В классе задаются свойства, константы и методы.
- Приватные свойства и методы доступны экземплярам одного и того же класса. Публичные свойства и методы доступны любому клиенту объекта.
- Объект является неизменяемым, если все его свойства нельзя изменять и если все объекты, содержащиеся в этих свойствах, сами являются неизменяемыми.
- Зависимости можно создавать на лету, получать из известного источника или передавать в конструктор в качестве аргумента (что называется *внедрением зависимостей*).
- При помощи наследования можно переопределять реализацию методов родительского класса. Интерфейс может объявлять методы, но их реализация будет производиться только в классах, которые будут использовать этот интерфейс.
- Полиморфизм означает, что код может использовать методы другого объекта согласно его типу (обычно это интерфейс), в результате чего поведение объекта будет соответствовать поведению экземпляра, предоставленного клиентом.
- Композиция — это назначение объекту другого объекта в качестве свойства.
- Модульные тесты проверяют поведение объектов.
- Во время тестирования можно заменять настоящие зависимости дублерами (например, заглушками mock и stub).
- Динамические массивы можно использовать для объявления списков и ассоциативных массивов без указания типов ключей и значений.

2 Создание сервисов

В этой главе

- ✓ Инстанцирование объектов-сервисов
- ✓ Внедрение и проверка зависимостей и значений конфигурации
- ✓ Преобразование необязательных аргументов конструктора в обязательные
- ✓ Преобразование неявных зависимостей в явные
- ✓ Проектирование неизменяемых сервисов

В следующих двух главах мы обсудим типы объектов и способы их инстанцирования. Если кратко, объекты бывают двух типов, и для каждого есть свои правила. В этой главе мы рассмотрим первый тип объектов: сервисы. Создание других объектов будет темой главы 3.

2.1. ДВА ТИПА ОБЪЕКТОВ

В приложении обычно присутствуют два типа объектов:

- объекты-сервисы, которые либо выполняют задачу, либо предоставляют информацию;

- объекты, которые содержат некоторые данные и, возможно, поведение для управления данными либо для их получения.

Объекты первого типа создаются единожды, а затем используются неоднократно, но в них самих ничего не меняется. У них очень простой жизненный цикл. После создания они могут существовать вечно, подобно механизмам, выполняющим определенные задачи. Эти объекты называются *сервисами*.

Объекты второго типа используются объектами первого типа для выполнения задач. Эти объекты выступают в качестве материала, с которым работают сервисы. Например, сервис может получить такой объект от другого сервиса, произвести с ним некоторые действия, а потом передать его для дальнейшей обработки (рис. 2.1). Жизненный цикл объекта-материала, таким образом, может выглядеть иначе, чем у сервиса: после создания с ним можно выполнять различные действия и даже хранить в нем журнал всех произошедших с ним событий.

Сервисные объекты — деятели, и их имена часто указывают на то, что они делают: controller, renderer, calculator. Сервисные объекты могут создаваться при помощи ключевого слова `new` при инстанцировании класса, например `new FileLogger()`.

В этой главе мы обсудим все особенности инстанцирования сервиса. Вы узнаете, как обращаться с зависимостями, о том, что можно, а что нельзя делать в конструкторе, а также как инстанцировать объект один раз и потом неоднократно использовать.

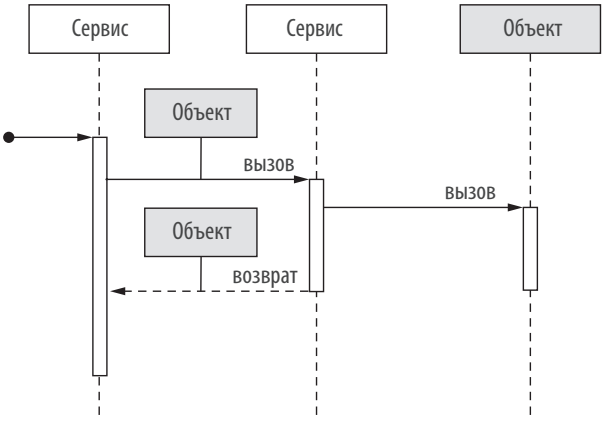


Рис. 2.1. На этой UML-диаграмме показано, как сервисы вызывают другие сервисы, передавая другие типы объектов в качестве аргументов методов или возвращаемых значений. В рамках метода сервиса над объектом можно производить манипуляции, или же сервис может получать из него данные

УПРАЖНЕНИЯ

1 Что из списка ниже является именем класса сервисного объекта?

- **а** `User`.
- **б** `EventDispatcher`.
- **в** `UserRepository`.
- **г** `Route`.

2 Что из списка ниже является именем класса другого объекта?

- **а** `DiscountCalculator`.
- **б** `Product`.
- **в** `TemplateRenderer`.
- **г** `Credentials`.

2.2. ВНЕДРЕНИЕ ЗАВИСИМОСТЕЙ И ЗНАЧЕНИЙ КОНФИГУРАЦИИ В КАЧЕСТВЕ АРГУМЕНТОВ КОНСТРУКТОРА

Чтобы выполнить задачу, сервис обычно нуждается в других сервисах. Эти другие сервисы — зависимости, и их нужно внедрять в качестве аргументов конструктора. Ниже представлен класс `FileLogger` как пример сервиса с зависимостью.

Листинг 2.1. Сервис FileLogger

```
interface Logger
{
    public function log(string message): void;
}

final class FileLogger implements Logger
{
    private Formatter formatter;

    public function __construct(Formatter formatter)
    {
        this.formatter = formatter;
    }

    public function log(string message): void
    {
        formattedMessage = this.formatter.format(message);
```

Formatter — зависимость класса FileLogger

```
        // ...
    }
}

logger = new FileLogger(new DefaultFormatter());
logger.log('A message');
```

Если предоставлять зависимости в качестве аргументов конструктора, то сервис можно будет использовать сразу после инстанцирования. Больше не нужно будет что-то указывать или настраивать, и вы никогда не забудете указать зависимость.

Бывает, что сервису нужны значения конфигурации, например путь для хранения файлов или входные данные для соединения с внешним сервисом. Внедряйте такие значения также в качестве аргументов конструктора, как показано ниже в листинге.

Листинг 2.2. Зависимости FileLogger требуется значение конфигурации

```
final class FileLogger implements Logger
{
    // ...

    private string logFilePath;

    public function __construct(
        Formatter formatter,
        string logFilePath
    ) {
        // ...

        this.logFilePath = logFilePath;
    }

    public function log(string message): void
    {
        // ...

        file_put_contents(
            this.logFilePath,
            formattedMessage,
            FILE_APPEND
        );
    }
}
```

logFilePath — значение конфигурации, которое передает FileLogger путь к файлу, где должны храниться сообщения

Эти значения конфигурации могут быть глобально доступными в приложении и располагаться в наборе параметров, конфигурационном объекте или в большой структуре данных вместе с другими значениями конфигурации. В таком случае вместо внедрения целого конфигурационного объекта можно внедрить лишь те значения, которые нужны сервису.

УПРАЖНЕНИЕ

3 Перепишите конструктор FileCache для получения только тех значений конфигурации, которые ему нужны, вместо того чтобы использовать целый конфигурационный объект приложения:

```
final class FileCache implements Cache
{
    private AppConfig appConfig;

    public function __construct(AppConfig appConfig)
    {
        this.appConfig = appConfig;
    }

    public function get(string cacheKey): string
    {
        directory = this.appConfig.get('cache.directory');

        // ...
    }
}
```

2.2.1. Хранение связанных файлов конфигурации

Внедрять весь глобальный конфигурационный объект не стоит, внедряйте только нужные значения. Тем не менее если некоторые из этих значений всегда будут использоваться вместе, то их внедрение по отдельности может нарушить их связность. Взгляните на следующий пример, где API-клиент получает входные данные для соединения с API, внедренным в качестве отдельного аргумента конструктора.

Листинг 2.3. Использование отдельных аргументов конструктора для имени и пароля пользователя

```
final class ApiClient
{
    private string username;
    private string password;

    public function __construct(string username, string password)
    {
        this.username = username;
        this.password = password;
    }
}
```

Чтобы передавать эти значения вместе, можно внедрить специальный конфигурационный объект. Вместо внедрения имени и пароля по отдельности, передайте объект `Credentials`, в котором будут содержаться оба значения.

Листинг 2.4. Имя пользователя и пароль теперь передаются вместе в объекте Credentials

```
final class Credentials
{
    private string username;
    private string password;

    public function __construct(string username, string password)
    {
        this.username = username;
        this.password = password;
    }

    public function username(): string
    {
        return this.username;
    }

    public function password(): string
    {
        return this.password;
    }
}

final class ApiClient
{
    private Credentials credentials;

    public function __construct(Credentials credentials)
    {
        this.credentials = credentials;
    }
}
```

УПРАЖНЕНИЕ

4 Перепишите конструктор класса `MySQLTableGateway` так, чтобы информация о соединении передавалась в качестве объекта.

```
final class MySQLTableGateway
{
    public function __construct(
        string host,
        int port,
        string username,
        string password,
        string database,
        string table
    ) {
        // ...
    }
}
```

2.3. ВНЕДРЯЙТЕ НЕОБХОДИМЫЕ СУЩНОСТИ, А НЕ МЕСТО ИХ РАСПОЛОЖЕНИЯ

Если рабочий фреймворк или библиотека достаточно сложны, в них можно найти особый тип объекта, содержащий любое необходимое сервисное или конфигурационное значение. Такой объект часто называют локатором, менеджером, реестром или контейнером сервисов.

ЧТО ТАКОЕ ЛОКАТОР СЕРВИСОВ?

Локатор сервисов — это тоже сервис, из которого можно извлекать другие сервисы. Ниже в примере показан локатор сервисов, в котором есть метод `get()`. При его вызове локатор вернет сервис с заданным идентификатором или выдаст исключение, если идентификатор распознается как недействительный.

Листинг 2.5. Упрощенная реализация локатора сервисов

```
final class ServiceLocator
{
    private array services;

    public function __construct()
    {
        this.services = [
            'logger' => new FileLogger(/* ... */)
        ];
    }

    public function get(string identifier): object
    {
        if (!isset(this.services[identifier])) {
            throw new LogicException(
                'Unknown service: ' . identifier
            );
        }

        return this.services[identifier];
    }
}
```

Здесь может быть любое количество сервисов

Таким образом, локатор сервисов — это своеобразный ассоциативный массив. Вы можете получать из него сервисы, если у вас есть правильный ключ. На практике ключом чаще всего служит имя нужного сервисного класса или интерфейса.

Реализация локатора сервисов обычно сложнее, чем в примере выше. Локатор сервисов зачастую располагает информацией о том, как инстанцировать все

сервисы приложения, и должен предоставлять необходимые аргументы для конструкторов в ходе инстанцирования. Он также может многократно использовать любые инстанцированные сервисы, что улучшает производительность среды исполнения.

Так как локатор сервисов предоставляет доступ ко всем сервисам приложения, может возникнуть соблазн внедрить локатор сервисов в качестве аргумента конструктора, как в листинге ниже, и покончить с этим.

Листинг 2.6. Использование ServiceLocator для получения зависимостей

```
final class HomepageController
{
    private ServiceLocator locator;

    public function __construct(ServiceLocator locator)
    {
        this.locator = locator;
    }

    public function execute(Request request): Response
    {
        user = this.locator.get(EntityManager.className)
            .getRepository(User.className)
            .getById(request.get('userId'));
        return this.locator.get(ResponseFactory.className)
            .create()
            .withContent(
                this.locator.get(TemplateRenderer.className)
                    .render(
                        'homepage.html.twig',
                        [
                            'user' => user
                        ]
                    ),
                'text/html'
            );
    }
}
```

Вместо внедрения необходимых зависимостей мы передаем целый ServiceLocator, из которого затем сможем получать любую зависимость

Это приводит к большому количеству дополнительных вызовов функций в коде, затмевая функцию `HomepageController`. Более того, поскольку сервисы не внедряются как зависимости, придется научить `HomepageController`, как их получать. И наконец, этот сервис теперь имеет доступ к широкому ряду других сервисов из локатора. В итоге локатор сервисов предоставляет массу несоответствующих

сущностей, так как такая структура не мотивирует программиста к вдумчивому проектированию.

Чтобы предотвратить все эти проблемы, руководствуйтесь следующим правилом: когда сервису необходим другой сервис для выполнения задачи, он должен явно объявить последний и внедрить его в качестве аргумента конструктора. `ServiceLocator` в предыдущем примере не является настоящей зависимостью `HomepageController`, он используется для *получения* настоящей зависимости. Поэтому вместо объявления `ServiceLocator` как зависимости контроллер должен объявить настоящую необходимую зависимость в качестве аргумента конструктора и ожидать ее внедрения.

Листинг 2.7. Внедрение настоящей зависимости в качестве аргумента конструктора

```
final class HomepageController
{
    private EntityManager entityManager;
    private ResponseFactory responseFactory;
    private TemplateRenderer templateRenderer;

    public function __construct(
        EntityManager entityManager,
        ResponseFactory responseFactory,
        TemplateRenderer templateRenderer
    ) {
        this.entityManager = entityManager;
        this.responseFactory = responseFactory;
        this.templateRenderer = templateRenderer;
    }

    public function execute(Request request): Response
    {
        user = this.entityManager.getRepository(User.className)
            .getById(request.get('userId'));

        return this.responseFactory
            .create()
            .withContent(
                this.templateRenderer.render(
                    'homepage.html.twig',
                    [
                        'user' => user
                    ]
                ),
                'text/html'
            );
    }
}
```

Итоговая диаграмма, показанная на рис. 2.2, лучше раскрывает ряд зависимостей класса.

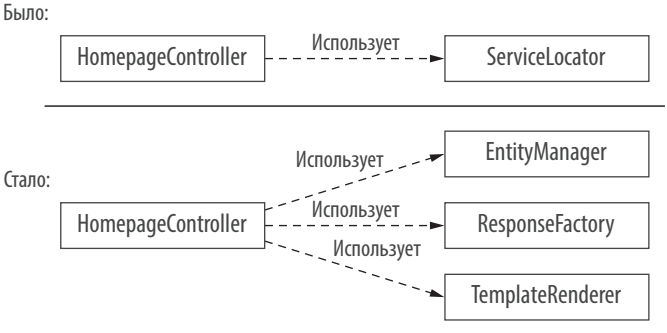


Рис. 2.2. В исходной версии казалось, что у HomepageController одна зависимость. После того как ServiceLocator убрали из зависимостей, видно, что на самом деле у HomepageController есть три зависимости

Добавим еще одну итерацию. Нам нужен `EntityManager` только для получения от него репозитория пользователя. Вместо этого надо сделать репозиторий явной зависимостью.

Листинг 2.8. Настоящая зависимость здесь UserRepository, а не EntityManager

```
final class HomepageController
{
    private UserRepository userRepository;
    // ...

    public function __construct(
        UserRepository userRepository,
        /* ... */
    ) {
        this.userRepository = userRepository
        // ...
    }

    public function execute(Request request): Response
    {
        user = this.userRepository
            .getById(request.get('userId'));
        // ...
    }
}
```

А ЧТО, ЕСЛИ НУЖЕН И САМ СЕРВИС, И СЕРВИС, ПОЛУЧАЕМЫЙ ОТ НЕГО?

Рассмотрите следующий код, в котором прописано использование зависимостей `EntityManager` и `UserRepository`:

```
user = this.entityManager
        .getRepository(User.className)
        .getById(request.get('userId'));
user.changePassword(newPassword);
this.entityManager.flush();
```

Если мы последуем совету и внедрим `UserRepository` вместо `EntityManager`, мы все равно вернемся к тому, что нам необходим `EntityManager` для очистки (добавления) его сущности.

В такой ситуации нужно перераспределять ответственность. Объект, который может получить сущность `User`, способен также хранить все изменения, которые в ней происходят. Фактически такие объекты будут следовать установленному паттерну — паттерну *репозитория*. Так как у нас уже есть класс `UserRepository`, имеет смысл добавить в него метод `flush()`, или (так как теперь можно задать ему другое имя) `save()`:

```
user = this.userRepository.getById(request.get('userId'));
user.changePassword(newPassword);
this.userRepository.save(user);
```

УПРАЖНЕНИЕ

5 Как убедиться, что вы внедряете нужную зависимость в конструктор сервисного объекта?

- **а** Если сервис вызывает метод при помощи этой зависимости.
- **б** Если сервис получает зависимость из нее.
- **в** Если сервис не получает зависимость из нее, а использует саму зависимость непосредственно.

2.4. ВСЕ АРГУМЕНТЫ КОНСТРУКТОРА ДОЛЖНЫ БЫТЬ ОБЯЗАТЕЛЬНЫМИ

Иногда зависимость бывает необязательной: объект может прекрасно функционировать и без нее. Пример такой необязательной зависимости — `Logger`, которую мы уже видели. Журналирование можно представить необязательной опцией при выполнении основной задачи.

Чтобы сделать зависимость сервиса необязательной, ее можно сделать необязательным аргументом конструктора, как в листинге ниже.

Листинг 2.9. Logger — необязательный аргумент конструктора

```
final class BankStatementImporter
{
    private Logger? logger;

    public function __construct(Logger? logger = null)
    {
        this.logger = logger;
    }

    public function import(string bankStatementFilePath): void
    {
        // Импорт файла выписки с банковского счета
        // Периодически записывайте лог информации для отладки ...
    }
}

importer = new BankStatementImporter();
```

logger может иметь значение null или ссылаться на экземпляр Logger

BankStatementImpropter можно инстанцировать без экземпляра Logger

Но это создает дополнительные сложности в коде класса `BankStatementImporter`. Когда вы хотите что-то журналировать, сначала нужно проверить, был ли предоставлен экземпляр `Logger` (иначе, если не проверите и `Logger` не будет предоставлен, возникнет фатальная ошибка):

```
public function import(string bankStatementFilePath): void
{
    // ...

    if (this.logger instanceof Logger) {
        this.logger.log('A message');
    }
}
```

Чтобы избавиться от лишней работы, лучше делать каждую зависимость обязательной. То же самое касается и значений конфигурации. Пользователь в `FileLogger` в принципе не обязан предоставлять путь к записи сообщений лога, если существует путь по умолчанию. Такое значение по умолчанию можно указать для соответствующего аргумента конструктора, как в примере ниже.

Листинг 2.10. Клиенту не обязательно предоставлять значение для logFilePath

```
final class FileLogger implements Logger
{
    public function __construct(
        string logFilePath = '/tmp/app.log'
    ) {
        // ...
```

```
    }
}

logger = new FileLogger();
```

Если пользователь не предоставит значение для аргумента logFilePath, то по умолчанию будет использоваться /tmp/app.log

Однако при инстанцировании `FileLogger` может быть неочевидно, куда записываются логи, особенно если значение по умолчанию будет спрятано глубоко в коде, как в примере ниже.

Листинг 2.11. Значение по умолчанию для logFilePath спрятано в методе log()

```
final class FileLogger implements Logger
{
    private string? logFilePath;

    public function __construct(string? logFilePath = null)
    {
        this.logFilePath = logFilePath;
    }

    public function log(string message): void
    {
        // ...

        file_put_contents(
            this.logFilePath != null ? this.logFilePath : '/tmp/app.log',
            formattedMessage,
            FILE_APPEND
        );
    }
}
```

Чтобы выявить путь, который будет использовать `FileLogger`, пользователю придется покопаться в коде и пересмотреть сам класс `FileLogger`. К тому же стандартный путь теперь является частью реализации класса, и его смену так просто не заметить. Вместо этого всегда запрашивайте у пользователя класса значения конфигурации для инициализации объектов. Если делать это при реализации всех классов, вы с легкостью сможете узнавать конфигурацию объектов лишь по тому, как они инстанцированы.

Итак, если аргументы конструктора используются для внедрения зависимостей или предоставления конфигурационных значений, то они должны быть обязательными и не содержать значений по умолчанию.

2.5. ВНЕДРЯЙТЕ ЗАВИСИМОСТИ ТОЛЬКО В КОНСТРУКТОРЕ

Еще один прием, иногда используемый для внедрения зависимостей, — добавление в класс метода-сеттера, который вызывается, если пользователь решит, что ему

нужна зависимость. Пример такого подхода показан в методе `setLogger()` класса `BankStatementImporter`, который позволяет клиенту внедрять в него сервис `Logger` после инстанцирования объекта.

Листинг 2.12. Logger можно предоставить позднее с вызовом setLogger()

```
final class BankStatementImporter
{
    private Logger? logger;

    public function __construct()
    {
    }

    public function setLogger(Logger logger): void
    {
        this.logger = logger;
    }

    // ...
}

importer = new BankStatementImporter();

importer.setLogger(logger);
```

Это решение создает проблему, о которой мы говорили ранее: оно усложняет код внутри класса. Более того, внедрение через сеттер нарушает два правила, которые мы еще обсудим:

- Объект нельзя создавать незавершенным.
- Сервисы должны быть неизменяемыми, а это значит, что их нельзя изменять после инстанцирования.

Итак, не внедряйте зависимости через сеттер, используйте конструктор.

2.6. НЕ СУЩЕСТВУЕТ НЕОБЯЗАТЕЛЬНЫХ ЗАВИСИМОСТЕЙ

Предыдущий раздел можно подытожить так: «Не существует необязательных зависимостей». Вам либо нужны зависимости, либо нет. Но, допустим, вы все еще рассматриваете журналирование в качестве вспомогательной функции. Как быть с учетом того, что я посоветовал использовать в конструкторе только обязательные аргументы? Во многих случаях вы решите эту проблему при помощи замещающего объекта, который выглядит как реальный, но ничего не делает, подобно представленной ниже реализации интерфейса `Logger` в классе `NullLogger`.

Листинг 2.13. Реализация интерфейса Logger ничего не делает

```
final class NullLogger implements Logger
{
    public function log(string message): void
    {
        // Ничего не делать
    }
}

importer = new BankStatementImporter(new NullLogger());
```

Такой безвредный объект часто называют *null object*, или *пустышкой*.

Если внедряемая зависимость является не сервисом, а некоторым значением конфигурации, делайте то же самое. Значение конфигурации должно быть обязательным аргументом, но, помимо этого, нужно предоставить пользователям возможность получать значение по умолчанию.

Листинг 2.14. Стандартный объект Configuration легко получить

```
final class MetadataFactory
{
    public function __construct(Configuration configuration)
    {
        // ...
    }
}

metadataFactory = new MetadataFactory(
    Configuration.createDefault()
);
```

Вместо того чтобы объявлять конфигурационные аргументы класса MetaDataFactory необязательными, предоставьте классу Configuration возможность получать значение по умолчанию

УПРАЖНЕНИЕ

6 У класса `CsvImporter` есть необязательная зависимость от объекта, который использует интерфейс `EventDispatcher`. Перепишите класс `CsvImporter` так, чтобы превратить `EventDispatcher` в обязательную зависимость. Предусмотрите удобную альтернативу для пользователей, которые не желают полностью реализовывать интерфейс `EventDispatcher`.

```
interface EventDispatcher
{
    public function dispatch(string eventName): void;
}

final class CsvImporter
{
    private EventDispatcher? eventDispatcher;
```

```
        public function __construct(EventDispatcher? eventDispatcher)
        {
            this.setEventDispatcher(eventDispatcher);
        }

        public function setEventDispatcher(
            EventDispatcher eventDispatcher
        ): void {
            this.eventDispatcher = eventDispatcher;
        }
    }
```

2.7. ДЕЛАЙТЕ ВСЕ ЗАВИСИМОСТИ ЯВНЫМИ

Даже если все зависимости и значения конфигурации должным образом внедряются в качестве аргументов конструктора, все равно нельзя исключать наличие *скрытых зависимостей*. Они называются скрытыми, так как их невозможно распознать, лишь взглянув на аргументы конструктора.

2.7.1. Преобразуйте статические зависимости в зависимости объектов

В некоторых приложениях можно получать глобально доступные зависимости при помощи статических методов доступа. Из любого места кода можно вызывать, например, `ServiceRegistry.get()` или `Cache.get()`. И если сервис извлекает зависимости таким образом, то перепишите его так, чтобы источником служили аргументы конструктора. Это сделает все зависимости явными.

Листинг 2.15. Внедрите экземпляр Cache вместо статических методов

```
// Было:
final class DashboardController
{
    public function execute(): Response
    {
        recentPosts = [];

        if (Cache.has('recent_posts')) {
            recentPosts = Cache.get('recent_posts');
        }

        // ...
    }
}

// Стало:
final class DashboardController
{
```

```
    private Cache cache;

    public function __construct(Cache cache)
    {
        this.cache = cache;
    }

    public function execute(): Response
    {
        recentPosts = [];

        if (this.cache.has('recent_posts')) {
            recentPosts = this.cache.get('recent_posts');
        }

        // ...
    }
}
```

2.7.2. Преобразуйте сложные функции в зависимости объектов

Иногда зависимости скрыты, потому что они являются функциями, а не объектами. Такие функции часто входят в стандартную библиотеку языка, например `json_encode()` или `simplexml_load_file()`, и в них спрятано множество зависимостей. Если писать код для этих функций самостоятельно, надо создавать множество классов и передавать их все в качестве зависимости в сервис. Таким образом создается настоящая объектная зависимость сервиса, а не скрытая статическая зависимость, чем по своей сути является функция.

Можно преобразовать функции в настоящие зависимости сервиса, написав класс-обертку для вызова функции. С его помощью в дальнейшем будет легче создавать любую логику на основе стандартных библиотечных функций, например для передачи аргументов по умолчанию или улучшения процессов обработки ошибок.

Листинг 2.16. JsonEncoder оборачивает вызов функции json_encode()

```
// Было:

final class ResponseFactory
{
    public function createApiResponse(array data): Response
    {
        return new Response(
            json_encode(data, JSON_THROW_ON_ERROR | JSON_FORCE_OBJECT),
            [
                'Content-Type' => 'application/json'
            ]
        );
    }
}
```

json_encode() — это скрытая зависимость

```
// Стало:

final class JsonEncoder
{
    /**
     * @throws RuntimeException
     */
    public function encode(array data): string
    {
        try {
            return json_encode(
                data,
                JSON_THROW_ON_ERROR | JSON_FORCE_OBJECT
            );
        } catch (RuntimeException previous) {
            throw new RuntimeException(
                'Failed to encode data: ' . var_export(data, true),
                0,
                previous
            );
        }
    }
}

final class ResponseFactory
{
    private JsonEncoder jsonEncoder;

    public function __construct(JsonEncoder jsonEncoder)
    {
        this.jsonEncoder = jsonEncoder;
    }

    public function createApiResponse(data): Response
    {
        return new Response(
            this.jsonEncoder.encode(data),
            [
                'Content-Type' => 'application/json'
            ]
        );
    }
}
```

С этого момента вызов json_encode() всегда будет с правильными аргументами

Теперь можно выдавать собственные исключения и получать больше информации для исправления ошибок

Экземпляр JsonEncoder теперь можно передавать в качестве актуальной и явной зависимости

Преобразование задачи обработки JSON-файлов в настоящую объектную зависимость класса `ResponseFactory` помогает пользователю понять назначение класса, взглянув лишь на аргументы конструктора. Впоследствии на основе объектной зависимости можно будет менять поведение сервиса, не переписывая его код. Мы обсудим это более подробно в главе 9.

ВСЕ ЛИ ФУНКЦИИ НЕОБХОДИМО ПРЕОБРАЗОВЫВАТЬ В ОБЪЕКТНЫЕ ЗАВИСИМОСТИ?

Не все функции нуждаются в оборачивании объектами, ряд функций можно передавать в качестве зависимостей как есть. Например, однострочные функции (`array_keys()`, `strpos()`) определенно не нужно оборачивать.

Чтобы определить, нужно ли создавать объектную зависимость, ответьте на следующие вопросы:

- Будете ли вы в дальнейшем менять или улучшать поведение класса посредством этой зависимости?
- Есть ли в поведении класса сложность, которую невозможно описать в нескольких строках кода?
- Работает ли функция с объектами вместо примитивных типов значений?

Если на большинство вопросов вы ответили «да», то определенно стоит превратить вызов функции в объектную зависимость. Дополнительным преимуществом станет то, что в тестах будет проще описывать ожидаемое поведение. Это поможет подставлять вместо вызова функции другие вызовы, пользовательский код или даже целую библиотеку, демонстрирующую аналогичное поведение.

2.7.3. Делайте вызовы системных функций явными

Ряд функций и классов, предоставляемых языком программирования, также стоит считать неявными зависимостями — это функции, которые получают данные из внешней среды. Например, класс `DateTime` и такие функции, как `time()` и `file_get_contents()`.

Рассмотрим следующий класс, `MeetupRepository`, который зависит от системных часов при получении текущего времени.

Листинг 2.17. MeetupRepository зависит от текущего времени

```
final class MeetupRepository
{
    private Connection connection;

    public function __construct(Connection connection)
    {
        this.connection = connection;
    }

    public function findUpcomingMeetups(string area): array
    {
        now = new DateTime();
```

При инстанцировании объекта DateTime без аргументов запрос текущего времени в системе будет осуществляться неявно

```
        return this.findMeetupsScheduledAfter(now, area);
    }

    public function findMeetupsScheduledAfter(
        DateTime time,
        string area
    ): array {
        // ...
    }
}
```

Текущее время — не та сущность, которую сервис может получить из предоставленных аргументов метода или из его зависимостей, поэтому это скрытая зависимость. И так как сервис, который может предоставлять текущее время, отсутствует, придется объявить собственный, как в листинге ниже.

Листинг 2.18. Интерфейс Clock можно использовать для получения текущего времени

```
interface Clock
{
    public function currentTime(): DateTime;
}

final class SystemClock implements Clock
{
    public function currentTime(): DateTime
    {
        return new DateTime();
    }
}

final class MeetupRepository
{
    // ...
    private Clock clock;

    public function __construct(
        Clock clock,
        /* ... */
    ) {
        this.clock = clock;
    }
    public function findUpcomingMeetups(string area): array
    {
        now = this.clock.currentTime();

        // ...
    }
}

meetupRepository = new MeetupRepository(new SystemClock());
meetupRepository.findUpcomingMeetups('NL');
```

Простое название для нового сервиса, который может передавать текущее время, — Clock (часы)

Стандартная реализация сервиса использует системное время для создания объекта DateTime, который содержит текущее время

Вместо создания объекта текущего времени на ходу теперь можно обращаться к сервису Clock

Перемещая вызов системной функции («Сколько сейчас времени?») за пределы класса `MeetupRepository`, мы улучшаем тестируемость самого класса `MeetupRepository`. Если запускать тесты в исходной ситуации, класс будет использовать фактическое текущее время. Результат в таком случае будет зависеть от даты и времени запуска теста. Это сделает тест нестабильным, и после определенной даты он завершится ошибкой. Если же использовать интерфейс Clock, то «текущее время» системных часов будет заменено «фиксированным временем», которое мы полностью контролируем, как показано ниже.

Листинг 2.19. Реализация интерфейса Clock для получения фиксированного значения текущего времени

```
final class FixedClock implements Clock
{
    private DateTime now;

    public function __construct(DateTime now)
    {
        this.now = now;
    }

    public function currentTime(): DateTime
    {
        return this.now;
    }
}

meetupRepository = new MeetupRepository(
    new FixedClock(
        new DateTime('2018-12-24 11:16:05')
    )
);
meetupRepository.findUpcomingMeetups('NL');
```

Реализацию интерфейса Clock в классе FixedClock можно использовать в тестах. При его инициализации нужно предоставить объект DateTime, который будет нести в себе значение текущего времени

При тестировании класса MeetupRepository объект FixedClock передается в качестве аргумента конструктора. Это сделает результаты теста однозначными

Передавая объект `Clock` в качестве аргумента конструктора, мы позволяем классу `MeetipRepository` запрашивать текущее время. Но также можно предоставлять возможность клиенту класса `MeetupRepository` передавать текущее время в качестве аргумента метода `findUpcomingMeetups()`. В этом случае отпадает зависимость от `Clock`.

Листинг 2.20. Текущее время можно передавать в качестве аргумента метода

```
final class MeetupRepository
{
    public function __construct(/* ... */)
    {
        // ...
    }
```

Зависимость Clock больше не нужна

```
    public function findUpcomingMeetups(
        string area,
        DateTime now  <-- Текущее время будет предоставляться клиентами этого метода
    ): array {
        // ...
    }
}
```

Сейчас стоит пересмотреть исходную идею о том, что для получения текущего времени нужно внедрять объектную зависимость. С передачей текущего времени в качестве аргумента метода конструктора такая информация превращается в контекст для поиска будущих событий встреч (митапов).

УПРАЖНЕНИЕ

7 UUID — произвольное число, которое можно использовать для ссылки на уникальные объекты в приложениях. Создание новых UUID зависит от системного генератора случайных чисел. Ниже представлен код для генерации UUID при помощи специального пакета:

```
final class CreateUser
{
    public function create(string username): void
    {
        userId = Uuid.create();

        user = new User(userId, username);
        // ...
    }
}
```

Что не так с этим кодом?

- **а** `Uuid` — это статическая зависимость, и ее нужно преобразовать в объектную зависимость.
- **б** Объект `Uuid` — зависимость сервиса, и нужно сделать так, чтобы он передавался в качестве аргумента конструктора.
- **в** `Uuid` — это конфигурационное значение, и оно должно передаваться в качестве аргумента конструктора.
- **г** `Uuid.create()` подразумевает вызов некой внешней сущности, поэтому его нужно создать в виде зависимости сервиса.

2.8. ПЕРЕДАВАЙТЕ ДАННЫЕ ДЛЯ ВЫПОЛНЕНИЯ ЗАДАЧ КАК АРГУМЕНТЫ МЕТОДА, А НЕ АРГУМЕНТЫ КОНСТРУКТОРА

Как вы знаете, сервис должен получать все зависимости и конфигурационные значения через аргументы конструктора. Но информация для выполнения самого задания, включая любую связанную с ним контекстуальную информацию, должна передаваться через аргументы метода.

В качестве контрпримера рассмотрим класс `EntityManager`, который можно использовать только для сохранения одной сущности в базе данных.

Листинг 2.21. Класс EntityManager, который можно использовать лишь для сохранения одного объекта

```
final class EntityManager
{
    private object entity;

    public function __construct(object entity)
    {
        this.entity = entity;
    }

    public function save(): void
    {
        // ...
    }
}

user = new User(/* ... */);
comment = new Comment(/* ... */);

entityManager = new EntityManager(user);
entityManager.save();

entityManager = new EntityManager(comment);
entityManager.save();
```

Чтобы сохранить еще одну сущность, придется инстанцировать еще один объект EntityManager

Это не очень полезный класс, так как его придется каждый раз инстанцировать заново.

Передача сущности в качестве аргумента конструктора — не самое подходящее решение в данном случае. Правильнее будет создать сервис, который получает текущий объект `Request` или `Session` в качестве аргумента конструктора.

Листинг 2.22. ContactRepository зависит от объекта Session

```
final class ContactRepository
{
    private Session session;
```

```
    public function __construct(Session session)
    {
        this.session = session;
    }

    public function getAllContacts(): array
    {
        return this.select()
            .where([
                'userId' => this.session.getUserId(),
                'companyId' => this.session.get('companyId')
            ])
            .getResult();
    }
}
```

Сервис `ContactRepository` находит контакты лишь тех пользователей или компаний, которые известны данному объекту `Session`. Таким образом, он способен работать только в одном контексте.

Передача части информации для задачи в виде аргументов конструктора ограничивает возможности для неоднократного использования сервиса и контекстуальной информации. Вся эта информация должна передаваться в виде аргументов методов, чтобы сервис можно было повторно использовать для других задач.

Чтобы решить, должна ли сущность передаваться в качестве аргумента конструктора или же метода, задайте себе такой вопрос: можно ли этот сервис вызвать для пакетной обработки, не прибегая к его повторному инстанцированию?. В зависимости от используемого языка сервис может вызываться один раз и использоваться повторно. Правда, если вы пишете на PHP, то любой инициализированный объект обычно существует лишь в течение времени, необходимого для обработки HTTP-запроса и возвращения ответа. В таком случае при проектировании сервисов задавайте вопрос иначе: если бы память не очищалась после каждого веб-запроса, можно ли использовать этот сервис для выполнения следующих запросов или же его пришлось бы инстанцировать заново?

Взгляните на сервис `EntityManager`, который мы рассматривали выше. Невозможно сохранить несколько сущностей за один раз без повторного инстанцирования сервиса. Поэтому `entity` нужно передавать как параметр метода `save()`, а не как аргумент конструктора.

Листинг 2.23. entity нужно передавать как аргумент метода

```
final class EntityManager
{
    public function save(object entity): void
    {
        // ...
    }
}
```

То же самое применимо к `ContactRepository`. Его нельзя использовать для получения контактных данных сразу нескольких пользователей или компаний. Методу `getAllContacts()` нужно передавать дополнительные аргументы по текущей компании или текущему пользователю, как показано ниже.

Листинг 2.24. UserId и CompanyId нужно передавать в качестве аргументов метода

```
final class ContactRepository
{
    public function getAllContacts(
        UserId userId,
        CompanyId companyId
    ): array {
        return this.select()
            .where([
                'userId' => userId,
                'companyId' => companyId
            ])
            .getResult();
    }
}
```

Кстати, слово «текущий» явно указывает на то, что это контекстуальная информация, и ее нужно передавать в качестве аргументов метода, например «текущее время», «идентификатор текущего пользователя», «текущий веб-запрос» и т. д.

УПРАЖНЕНИЕ

8 Что не так с кодом ниже?

```
final Translator
{
    private string userLanguage;

    public function __construct(string userLanguage)
    {
        this.userLanguage = userLanguage;
    }

    public function translate(string messageKey): string
    {
        // ...
    }
}
```

a Аргумент конструктора `userLanguage` должен иметь значение по умолчанию, если на момент инстанцирования пользователь не вошел в систему.

б Значение языка текущего пользователя должно извлекаться из сервиса, передаваемого в качестве аргумента конструктора.

в `userLanguage` должен передаваться посредством аргумента при вызове метода `translate()`.

г Передавая `userLanguage` в качестве аргумента конструктора, мы затрудняем повторное использование сервиса `Translator`.

2.9. ПОСЛЕ ИНСТАНЦИРОВАНИЯ СЕРВИСА ЕГО ПОВЕДЕНИЕ ДОЛЖНО ОСТАВАТЬСЯ НЕИЗМЕННЫМ

Как мы уже говорили, объявляя в сервисе необязательные зависимости после его инстанцирования, вы изменяете его поведение. Это делает сервис непредсказуемым. То же самое справедливо для методов, которые не используют зависимости, но позволяют влиять на поведение сервиса извне. В качестве примера рассмотрим метод `ignoreErrors()` класса `Importer` в листинге ниже.

Листинг 2.25. При вызове ignoreErrors() меняется поведение объекта Importer

```
final class Importer
{
    private bool ignoreErrors = true;

    public function ignoreErrors(bool ignoreErrors): void
    {
        this.ignoreErrors = ignoreErrors;
    }

    // ...
}

importer = new Importer();

// ...

importer.ignoreErrors(false);

// ...
```

Когда мы используем Importer, ошибки будут игнорироваться

Здесь же ошибки игнорироваться не будут

Убедитесь, что этого не произойдет. Все зависимости и конфигурационные значения должны быть заданы с самого начала; необходимо убрать возможность перенастройки сервиса после его инстанцирования.

Еще один пример — класс `EventDispatcher` в следующем листинге. Он позволяет перенастраивать список активных прослушивателей событий после инстанцирования.

Листинг 2.26. Поведение EventDispatcher может меняться после инстанцирования

```
final class EventDispatcher
{
    private array listeners = [];

    public function addListener(
        string event,
        callable listener
    ): void {
        this.listeners[event][] = listener;
    }

    public function removeListener(
        string event,
        callable listener
    ): void {
        foreach (this.listenersFor(event) as key => callable) {
            if (callable == listener) {
                unset(this.listeners[event][key]);
            }
        }
    }

    public function dispatch(object event): void
    {
        foreach (this.listenersFor(event.className) as callable) {
            callable(event);
        }
    }

    private function listenersFor(string event): array
    {
        if (isset(this.listeners[event])) {
            return this.listeners[event];
        }

        return [];
    }
}
```

Можно добавить новый прослушиватель для заданного типа событий

Также можно удалить существующий прослушиватель

Каждый существующий на данный момент прослушиватель будет вызван

Позволяя добавлять и удалять прослушиватели событий на лету, мы делаем поведение сервиса `EventDispatcher` непредсказуемым, так как оно теперь может меняться со временем. В данном случае следует превратить массив прослушивателей

в аргумент конструктора и удалить методы `addListener()` и `removeListener()`, как это сделано в листинге ниже.

Листинг 2.27. Прослушиватели событий могут настраиваться только при создании

```
final class EventDispatcher
{
    private array listeners;

    public function __construct(array listenersByEventName)
    {
        this.listeners = listenersByEventName;
    }

    // ...
}
```

«В МОЕМ ПРИЛОЖЕНИИ НУЖНЫ ИЗМЕНЯЕМЫЕ СЕРВИСЫ»

Хорошее замечание. Сам я занимался в основном разработкой веб-приложений, и, по моему опыту, в них не нужны изменяемые сервисы, так как все необходимое поведение всегда можно задать во время вызова конструктора.

Вы, возможно, работаете над другими типами приложений, где нужен сервис наподобие диспетчера событий, который позволяет добавлять или удалять прослушиватели или подписчиков после вызова конструктора. Например, если вы создаете игру или другое интерактивное приложение с пользовательским интерфейсом и пользователь открывает новое окно, нужно будет создавать новые прослушиватели событий для UI-элементов окна. Позже, после закрытия окна, нужно будет удалять эти прослушиватели. В таких случаях сервисам и вправду нужно быть изменяемыми. Я лишь призываю вас подумать, как избежать ситуации, когда одни объекты перенастраивают поведение других посредством таких публичных методов, как `addListener()` и `removeListener()`.

Поскольку массив — это не специфичный тип, который может содержать в себе все что угодно (если вы пишете на динамически типизированном языке), стоит проверять аргументы для `listenersByEventName` перед передачей. Позже в этой главе мы рассмотрим проверку аргументов конструктора.

Без разрешения изменяться после инстанцирования сервис также не получит необязательные зависимости и будет вести себя предсказуемо, не выбирая иные пути исполнения, зависящие от того, кто вызывает метод (рис. 2.3).

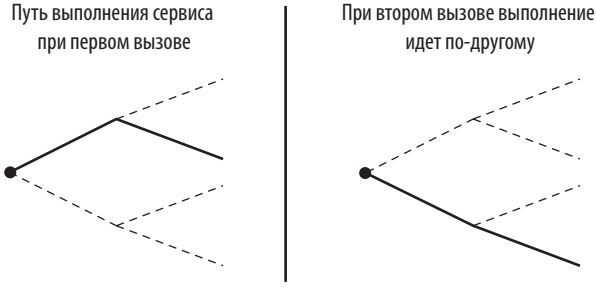


Рис. 2.3. Разрешение изменять поведение сервиса после инстанцирования приводит к тому, что сервис ведет себя непредсказуемо для некоторых клиентов, поскольку может внезапно выбрать другой путь выполнения

2.10. НЕ ДЕЛАЙТЕ НИЧЕГО ВНУТРИ КОНСТРУКТОРА, КРОМЕ ИНИЦИАЛИЗАЦИИ СВОЙСТВ

Создание сервиса подразумевает передачу аргументов его конструктору, что подготовит сервис к использованию. Сама полезная работа сервиса будет осуществляться методами инстанцированного объекта. Иногда возникает соблазн сделать внутри конструктора что-то большее, чем просто инициализация свойств, чтобы еще лучше подготовить объект к использованию. Например, рассмотрим следующий класс `FileLogger`. При вызове конструктора он подготавливает файл журнала для записи.

Листинг 2.28. При необходимости FileLogger создает папку для файлов журнала

```
final class FileLogger implements Logger
{
    private string logFilePath;

    public function __construct(string logFilePath)
    {
        logFileDirectory = dirname(logFilePath);
        if (!is_dir(logFileDirectory)) {
            mkdir(logFileDirectory, 0777, true);
        }

        touch(logFilePath);

        this.logFilePath = logFilePath;
    }

    // ...
}
```

Создает папку, если ее еще не существует

При инстанцировании класса `FileLogger` мы затрагиваем саму систему, даже если никогда не воспользуемся объектом для записи в журнале.

Признак хорошего тона — не делать ничего в конструкторе кроме того, что необходимо. А все, что необходимо, — это проверять предоставленные аргументы, а затем присваивать их в качестве значения для свойств сервиса.

Листинг 2.29. Конструктор класса FileLogger теперь не создает папок

```
final class FileLogger implements Logger
{
    private string logFilePath;

    public function __construct(string logFilePath)
    {
        this.logFilePath = logFilePath;
    }

    public function log(string message): void
    {
        this.ensureLogFileExists();

        // ...
    }

    private function ensureLogFileExists(): void
    {
        if (is_file(this.logFilePath)) {
            return;
        }

        logFileDirectory = dirname(this.logFilePath);
        if (!is_dir(logFileDirectory)) {
            mkdir(logFileDirectory, 0777, true);
        }

        touch(this.logFilePath);
    }
}
```

Всего лишь копирует значения для свойств

Можно переместить полезную нагрузку из конструктора внутрь класса. Тем не менее в таком случае мы узнаем, есть ли возможность записывать сообщения в журнал, только при попытке добавления в него первой записи. Можно перенести выполнение этой задачи за пределы конструктора, чтобы узнать о возможной проблеме с записью заранее, до вызова `FileLogger`, а не после этого. Возможно, тут окажется полезен сервис `LoggerFactory`.

Листинг 2.30. LoggerFactory создаст папку для файла журнала

```
final class FileLogger implements Logger
{
    private string logFilePath;

    public function __construct(string logFilePath)
    {
        if (!is_writable(logFilePath)) {
```

Мы ожидаем, что путь для файла журнала верен, поэтому здесь проводим только проверку безопасности

```
            throw new InvalidArgumentException(
                'Log file path "{logFilePath}" should be writable'
            );
        }
        this.logFilePath = logFilePath;
    }

    public function log(string message): void
    {
        // ...
    }
}

final class LoggerFactory
{
    public function createFileLogger(string logFilePath): FileLogger
    {
        if (!is_file(logFilePath)) {
            logFileDirectory = dirname(logFilePath);
            if (!is_dir(logFileDirectory)) {
                mkdir(logFileDirectory, 0777, true);
            }

            touch(logFilePath);
        }

        return new FileLogger(logFilePath);
    }
}
```

Нет необходимости проверять, что файл уже существует, и т. д.

Задача создания папки и файла журнала теперь передана фазе инициализации самого приложения

Заметьте, что перемещение конфигурационного кода за пределы конструктора сервиса `FileLogger` меняет условия работы самого `FileLogger`. Изначально вы могли передавать любой путь к файлу журнала, и `FileLogger` делал всю остальную работу (создавал папку при необходимости и проверял, что сам путь еще актуален). Сейчас же `FileLogger` принимает путь к файлу журнала и ожидает, что нужная папка уже существует. Можно переместить в фазу инициализации еще больше работы и переписать `FileLogger` так, чтобы клиент предоставлял путь к файлу, который уже существует и доступен для записи. В листинге ниже показано, как это будет выглядеть.

Листинг 2.31. LoggerFactory предоставляет все, что необходимо сервису FileLogger

```
final class FileLogger implements Logger
{
    private string logFilePath;

    /**
     * @param string logFilePath Полный путь к файлу журнала,
     * который уже существует и доступен для записи.
     */
```

```
    public function __construct(string logFilePath)
    {
        this.logFilePath = logFilePath;
    }

    // ...
}

final class LoggerFactory
{
    public function createFileLogger(string logFilePath): FileLogger
    {
        if (!is_file(logFilePath)) {
            logFileDirectory = dirname(logFilePath);
            if (!is_dir(logFileDirectory)) {
                mkdir(logFileDirectory, 0777, true);
            }

            touch(logFilePath);
        }

        if (!is_writable(logFilePath)) {
            throw new InvalidArgumentException(
                'Log file path "{logFilePath}" should be writable'
            );
        }

        return new FileLogger(logFilePath);
    }
}
```

Кроме создания папки, LoggerFactory теперь еще проверяет, что файл журнала существует и доступен для записи

Рассмотрим еще один, менее очевидный пример объекта, который выполняет операции в конструкторе. Взгляните на следующий класс `Mailer`, который обращается к одной из своих зависимостей при вызове конструктора.

Листинг 2.32. Mailer осуществляет некоторые действия в конструкторе

```
final class Mailer
{
    private Translator translator;
    private string defaultSubject;

    public function __construct(Translator translator)
    {
        this.translator = translator;

        // ...

        this.defaultSubject = this.translator
            .translate('default_subject');

    }
    // ...
}
```

Но что произойдет, если изменить порядок присваивания значений свойствам?

Листинг 2.33. Изменение порядка присваивания значений в конструкторе класса Mailer

```
final class Mailer
{
    private Translator translator;
    private string defaultSubject;

    public function __construct(
        Translator translator,
        string locale
    ) {
        this.defaultSubject = this.translator
            .translate('default_subject', locale);

        // ...

        this.translator = translator;
    }

    // ...
}
```

Теперь вы получите фатальную ошибку из-за вызова `translate()` с `null`. Вот почему правило, согласно которому присваивать значения свойствам можно только в конструкторе, приводит к тому, что присвоения могут выполняться в любом порядке. Если присвоения должны выполняться в определенном порядке, вы знаете, что выполняете операции внутри конструктора.

Конструктор класса `Mailer` также является примером того, как контекстуальная информация, в данном случае локаль конкретного пользователя, иногда передается в качестве аргумента конструктора. Как вы знаете, правильнее передавать контекстуальные данные в качестве аргументов методов.

УПРАЖНЕНИЕ

9 Взгляните на класс `MySQLTableGateway`. Он подключается к базе данных при помощи объекта `ConnectionConfiguration`.

```
final class MySQLTableGateway
{
    private Connection connection;

    public function __construct(
        ConnectionConfiguration connectionConfiguration,
        string tableName
    ) {
```

```
            this.tableName = tableName;
            this.connect(connectionConfiguration);
        }

        private function connect(
            ConnectionConfiguration connectionConfiguration
        ): void {
            this.connection = new Connection(
                // ...
            );
        }

        public function insert(array data): void
        {
            this.connection.insert(this.tableName, data);
        }
    }
```

Перепишите этот класс, чтобы убедиться, что конструктор не делает ничего, кроме присвоения значений свойствам.

2.11. ВЫДАВАЙТЕ ИСКЛЮЧЕНИЕ ПРИ НЕДОПУСТИМОМ АРГУМЕНТЕ

Когда клиент класса предоставляет недопустимый аргумент конструктора, средство проверки типов выдает предупреждение (например, что для аргумента требуется экземпляр `Logger`, а клиент передает значение `bool`). Тем не менее существуют другие типы аргументов, при работе с которыми недостаточно полагаться лишь на систему проверки типов. Например, в следующем классе `Alerting` один из аргументов конструктора должен быть `int` — это конфигурационный флаг.

Листинг 2.34. Класс Alerting ожидает аргумент конструктора типа int

```
final class Alerting
{
    private int minimumLevel;

    public function __construct(int minimumLevel)
    {
        this.minimumLevel = minimumLevel;
    }
}

alerting = new Alerting(-99999999);
```

Если принимать для свойства `minimumLevel` любое `int`-значение, нет гарантии, что это значение будет реалистичным и его можно будет использовать в остальной

части кода. Значения надо проверять на допустимость в конструкторе, и если они не проходят проверку, выдавать исключение. Значение будет присвоено свойству только после проверки аргумента, как показано ниже.

Листинг 2.35. Проверка перед присвоением значения свойству

```
final class Alerting
{
    private int minimumLevel;

    public function __construct(int minimumLevel)
    {
        if (minimumLevel <= 0) {
            throw new InvalidArgumentException(
                'Minimum alerting level should be greater than 0'
            );
        }
        this.minimumLevel = minimumLevel;
    }
}

alerting = new Alerting(-99999999); ◄── Это спровоцирует исключение InvalidArgumentException
```

Выдавая исключение в пределах конструктора, вы предотвратите создание объекта с недопустимыми аргументами.

> **ПРИМЕЧАНИЕ** Вместо того чтобы выдавать собственные исключения, можно использовать готовые функции проверки аргументов конструкторов и методов. Мы поговорим об этом подробнее в разделе 3.7.

Можно обойтись и без выдачи исключений, если это не повлияет на дальнейшее поведение объекта. Рассмотрим следующий класс `Router`.

Листинг 2.36. Router не выдает исключений

```
final class Router
{
    private array controllers;
    private string notFoundController;

    public function __construct(
        array controllers,
        string notFoundController
    ) {
        this.controllers = controllers;
        this.notFoundController = notFoundController;
    }
```

Нужно ли проверять, что массив контроллеров не пустой?

```
    public function match(string uri): string
    {
        foreach (this.controllers as pattern => controller) {
            if (this.matches(uri, pattern)) {
                return controller;
            }
        }

        return this.notFoundController;
    }

    private function matches(string uri, string pattern): bool
    {
        // ...
    }
}

router = new Router(
    [
        '/' => 'homepage_controller'
    ],
    'not-found'
);

router.match('/');  ◄—— Здесь будет возвращен homepage_controller
```

Нужно ли здесь проверять аргумент `controllers`, чтобы убедиться, что в нем содержится хотя бы одна пара имен URI паттернов и контроллеров? Не нужно, так как поведение объекта `Router` не будет нарушено, если массив `controllers` окажется пустым. Если массив пуст и клиент вызовет функцию `match()`, то в результате вернется лишь «controller not found» (контроллер не найден), так как не найдено подходящих паттернов для данного URI (и других URI). Такое поведение мы ожидаем от маршрутизатора, поэтому не рассматриваем его в качестве признака заведомо неверной логики.

Тем не менее стоит проверять, что все ключи и значения в массиве `controllers` являются строками. Это поможет заранее выявлять ошибки программирования. Рассмотрите следующий пример.

Листинг 2.37. Router должен проверять массив controllers

```
final class Router
{
    // ...

    public function __construct(array controllers)
    {
        foreach (array_keys(controllers) as pattern) {
```

```
            if (!is_string(pattern)) {
                throw new InvalidArgumentException(
                    'Все паттерны URI должны представлять собой строки'
                );
            }
        }
        foreach (controllers as controller) {
            if (!is_string(controller)) {
                throw new InvalidArgumentException(
                    'Все контроллеры должны представлять собой строки'
                );
            }
        }
        this.controllers = controllers;
    }

    // ...
}
```

Для проверки содержания массива `controllers` можно также использовать библиотеки и пользовательские функции утверждений (мы обсудим функции проверки утверждений подробнее в разделе 3.7) или позволить системе типов проводить проверку, как в листинге ниже. Так как метод `addController()` имеет явно указанный тип строки в аргументах, то вызов этого метода для каждой пары «ключ — значение» в массиве контроллеров будет эквивалентен утверждению, что все ключи и значения массива являются строками.

Листинг 2.38. Альтернативный способ проверки элементов массива controllers

```
final class Router
{
    private array controllers = [];

    public function __construct(array controllers)
    {
        foreach (controllers as pattern => controller) {
            this.addController(pattern, controller);
        }
    }

    private function addController(
        string pattern,
        string controller
    ): void {
        this.controllers[pattern] = controller;
    }

    // ...
}
```

Не назначайте контроллеры напрямую, пусть addController() делает это за вас

УПРАЖНЕНИЕ

10 Конструктор класса `EventDispatcher` не проверяет должным образом, имеет ли предоставленный аргумент `eventListeners` обязательную структуру. Перепишите конструктор так, чтобы выдавалось исключение, когда клиент передает недопустимое значение.

```
final class EventDispatcher
{
    private array eventListeners;

    public function __construct(array eventListeners)
    {
        this.eventListeners = eventListeners;
    }

    public function dispatch(object event): void
    {
        eventName = event.className;

        listeners = isset(this.eventListeners[eventName]) ?
            this.eventListeners[eventName] : [];

        foreach (listeners as listener) {
            listener(event);
        }
    }
}
```

2.12. ОБЪЯВЛЯЙТЕ СЕРВИСЫ КАК НЕИЗМЕНЯЕМЫЕ ГРАФЫ ОБЪЕКТОВ С ОГРАНИЧЕННЫМ РЯДОМ ТОЧЕК ВХОДА

Когда фреймворк приложения вызывает контроллер (будь это веб-контроллер или контроллер для консольного приложения), уже можно считать все зависимости известными. Например, веб-контроллеру необходим репозиторий, из которого он будет получать объекты, необходим механизм шаблонов для рендеринга шаблона, а также фабрика ответов для создания объекта ответа и т. д. У всех этих зависимостей есть свои зависимости, перечисленные в ряде аргументов конструкторов, и в итоге мы имеем зачастую очень большие графы объектов.

Если фреймворку потребуется вызвать другой контроллер, он будет использовать еще один граф объектов. Сам контроллер также является сервисом со своими зависимостями, поэтому контроллеры можно рассматривать как точки входа для графа объектов в приложении, как показано на рис. 2.4.

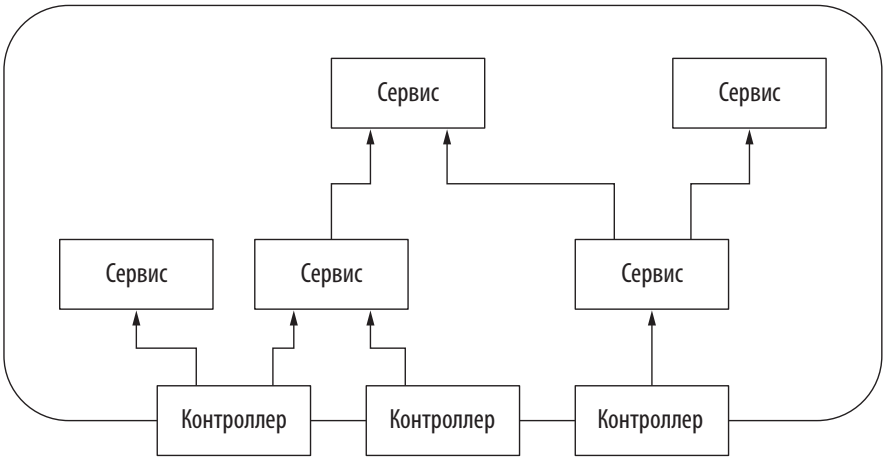


Рис. 2.4. Граф содержит все сервисы приложения, причем сервисы контроллера помечены как сервисы точки входа. Это единственные сервисы, которые можно получить напрямую; все остальные сервисы доступны только как внедренные зависимости

У большинства приложений есть нечто наподобие сервисного контейнера, который указывает, как можно создавать все сервисы приложения, какие у них зависимости, как вызывается их конструктор и т. д. Контейнер также ведет себя как локатор сервиса. Можно запросить у него один из сервисов для дальнейшего использования. Мы уже встречались с локаторами сервисов в разделе 2.3, когда обсуждали правило: нужно передавать только необходимые зависимости, а не локатор, который позволяет получать эти зависимости.

Исходя из того что:

- все сервисы в приложении составляют один большой граф объектов;
- контроллеры будут являться точками входа;
- никакому сервису не потребуется локатор для получения сервиса,

делаем вывод: контейнеру сервисов нужно предоставлять лишь публичные методы для получения контроллеров. Другие сервисы, содержащиеся в контейнере, могут и должны оставаться приватными, так как они будут использоваться лишь в качестве передаваемых зависимостей для контроллеров.

На языке кода это значит, что можно использовать контейнер сервисов как локатор сервисов, чтобы получать из него контроллеры. Вся остальная логика

инициализации, необходимая для создания объектов-контроллеров, остается за кадром, в приватных методах.

Листинг 2.39. Публичные методы предназначены для точек входа, приватные — для зависимостей

```
final class ServiceContainer
{
    public function homepageController(): HomepageController
    {
        return new HomepageController(
            this.userRepository(),
            this.responseFactory(),
            this.templateRenderer()
        );
    }

    private function userRepository(): UserRepository
    {
        // ...
    }

    private function responseFactory(): ResponseFactory
    {
        // ...
    }

    private function templateRenderer(): TemplateRenderer
    {
        // ...
    }

    // ...
}

if (uri == '/') {
    controller = serviceContainer.homepageController();
    response = controller.execute(request);
    // ...
} elseif (/* ... */) {
    // ...
}
```

Фреймворк может использовать маршрутизатор, чтобы искать контроллер для данного запроса. Затем он получает контроллер от локатора сервисов и делегирует ему выполнение запроса

Получить ответ и вызвать другой контроллер

Контейнер сервисов позволяет использовать сервисы повторно, поэтому, начиная с контроллера в качестве точки входа, не каждая ветвь графа объектов будет полностью автономной. Например, другой контроллер может использовать тот же экземпляр `TemplateRenderer`, что и `HomepageController` (рис. 2.5). Именно поэтому важно делать сервисы максимально предсказуемыми. Если применить к ним все ранее упомянутые правила, получим граф объектов, который можно инстанцировать единожды и использовать многократно.

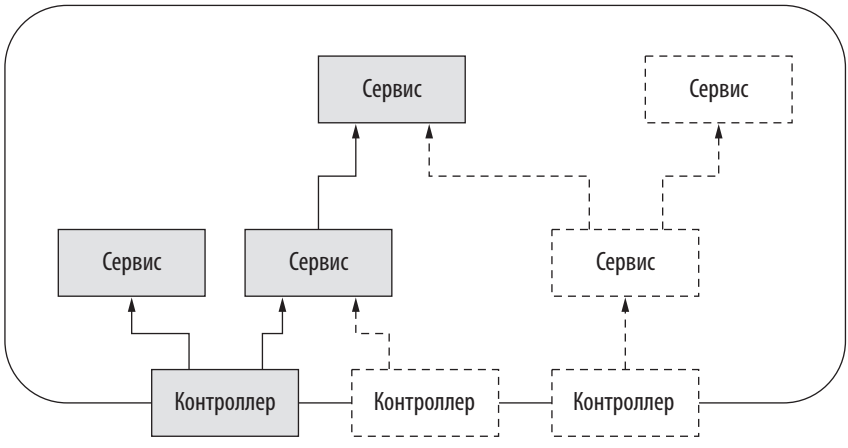


Рис. 2.5. Различные входные точки используют различные ветви графа объектов

ЗАКЛЮЧЕНИЕ

- Сервисы должны создаваться в одно действие с передачей зависимостей и значений конфигурации в качестве аргументов конструктора. Все зависимости сервиса должны быть явными и передаваться как объекты. Все значения конфигурации должны проверяться. Когда конструктор получает недопустимый аргумент, должно выдаваться исключение.
- После вызова конструктора сервис должен быть неизменяемым, его поведение не должно меняться при вызове любых его методов.
- Все сервисы приложения совместно формируют большой неизменяемый граф объектов, который зачастую управляется контейнером сервисов. Контроллеры являются точками входа в эти графы. Сервисы можно инстанцировать один раз и использовать неоднократно.

ОТВЕТЫ К УПРАЖНЕНИЯМ

1 Правильные ответы: **б** и **в**.

2 Правильные ответы: **б** и **г**.

3 Вариант ответа:

```
final class FileCache implements Cache
{
    private string cacheDirectory;
```

```
    public function __construct(string cacheDirectory)
    {
        this.cacheDirectory = cacheDirectory;
    }

    // ...
}
```

4 Вариант ответа:

```
final class MySQLTableGateway
{
    public function __construct(
        ConnectionConfiguration connectionConfiguration,
        string table
    ) {
        // ...
    }
}
```

Имя таблицы не является информацией, необходимой для соединения с базой данных, поэтому мы не передаем его в объект ConnectionConfiguration

5 Правильный ответ: **в**. Передаваемая зависимость должна использоваться непосредственно, а не для получения настоящей зависимости.

6 Для начала сделайте `eventDispatcher` обязательным аргументом и удалите метод `setEventDispatcher()`:

```
final class CsvImporter
{
    private EventDispatcher eventDispatcher;

    public function __construct(EventDispatcher eventDispatcher)
    {
        this.eventDispatcher = eventDispatcher;
    }
}
```

Затем передайте «болванку» `EventDispatcher`, которую клиенты могут использовать, если им не требуется управления событиями:

```
final class EventDispatcherDummy implements EventDispatcher
{
    public function dispatch(string eventName): void
    {
        // ничего не делать
    }
}
```

7 Правильный ответ: **г**. Хотя `Uuid.create()` — это статический метод, он не является статической зависимостью, которую можно передавать в качестве аргумента конструктора (по сути, это именованный конструктор). `Uuid` также не является значением конфигурации, так как его текущее

значение будет уникальным при каждом создании нового экземпляра после вызова `Uuid.create()`.

8 Правильные ответы: **в** и **г**. Язык пользователя в данном случае — контекстуальная информация, которую нужно передавать через аргументы методов. Она не должна передаваться через аргументы конструктора или запрашиваться от внедряемого сервиса. Если передать ее в `translator`, мы также избежим зависимости сервиса от неявной контекстуальной информации.

9 Вариант ответа:

```
final class MySQLTableGateway
{
    private ConnectionConfiguration connectionConfiguration;

    public function __construct(
        ConnectionConfiguration connectionConfiguration,
        string tableName
    ) {
        this.connectionConfiguration = connectionConfiguration;
        this.tableName = tableName;
    }

    private function connect(): void
    {
        if (this.connection instanceof Connection) {
            return;
        }

        this.connection = new Connection(
            // ...
        );
    }

    public function insert(array data): void
    {
        this.connect();

        this.connection.insert(this.tableName, data);
    }
}
```

Храните конфигурацию соединения в свойстве, чтобы в дальнейшем его можно было использовать для соединения с базой данных

Проверьте, не установлено ли уже соединение

Используйте this.connectionConfiguration для настройки текущего соединения

Когда бы ни понадобилось соединение, сначала вызывается метод connect()

10 Вариант ответа:

```
public function __construct(array eventListeners)
{
    foreach (eventListeners as eventName => listeners) {
        if (!is_string(eventName)) {
            throw new InvalidArgumentException(
                'eventName должен быть строкой'
            );
```

```
            }
            if (!is_array(listeners)) {
                throw new InvalidArgumentException(
                    'Прослушиватель должен быть массивом'
                );
            }
            foreach (listeners as listener) {
                if (!is_callable(listener)) {
                    throw new InvalidArgumentException(
                        'Прослушиватель должен быть вызываемым'
                    );
                }
            }
        }

        this.eventListeners = eventListeners;
    }
```

Вот другой вариант, в котором используется проверка типов:

```
private array eventListeners = [];

    public function __construct(array eventListeners)
    {
        foreach (eventListeners as eventName => listeners) {
            this.addListeners(eventName, listeners);
        }
    }

    private function addListener(string eventName, array listeners): void
    {
        foreach (listeners as listener) {
            if (!is_callable(listener)) {
                throw new InvalidArgumentException(
                    'Прослушиватель должен быть вызываемым'
                );
            }
        }

        this.eventListeners[eventName] = listeners;
    }
```

Здесь постепенно собираются значения для свойства eventListeners, поэтому его нужно инициализировать пустым массивом

Типы параметров гарантируют, что ключи каждого значения в исходном массиве eventListeners являются строками, а также что соответствующие значения являются массивами

3 Создание других объектов

В этой главе:

- ✓ Инстанцирование других типов объектов
- ✓ Предотвращение незавершенного состояния объектов
- ✓ Сохранение инвариантности предметной области
- ✓ Использование именованных конструкторов
- ✓ Использование проверок утверждений

Ранее я упоминал, что существуют два типа объектов: сервисы и все остальные объекты. Второй тип можно разделить на более специфичные подтипы, а именно *объекты-значения* и *сущности* (иногда называемые моделями). Сервисы будут создавать или получать сущности, выполнять с ними операции либо передавать их другим сервисам. Они также будут создавать объекты-значения и передавать их в качестве аргументов методов или создавать их модифицированные копии. В некотором смысле сущности и объекты-значения являются материалом, который сервисы используют для выполнения задач.

В главе 2 мы рассмотрели, как создавать объекты-сервисы. В этой главе познакомимся с правилами создания других объектов.

3.1. ЗАПРАШИВАЙТЕ МИНИМАЛЬНО НЕОБХОДИМЫЙ ДЛЯ СОГЛАСОВАННОГО ПОВЕДЕНИЯ ОБЪЕКТА ОБЪЕМ ДАННЫХ

Взгляните на следующий класс `Position`.

Листинг 3.1. Класс Position

```
final class Position
{
    private int x;
    private int y;

    public function __construct()
    {
        // пусто
    }

    public function setX(int x): void
    {
        this.x = x;
    }

    public function setY(int y): void
    {
        this.y = y;
    }

    public function distanceTo(Position other): float
    {
        return sqrt(
            (other.x - this.x) ** 2 +
            (other.y - this.y) ** 2
        );
    }
}

position = new Position();
position.setX(45);
position.setY(60);
```

Пока не вызваны методы `setX()` и `setY()`, объект находится в несогласованном состоянии. Это можно заметить, если вызвать метод `distanceGo()` до вызова `setX()` и `setY()`; в этом случае вы не получите значимого ответа.

Поскольку для концепции расположения важно иметь значения `x` и `y`, необходимо сделать так, чтобы создать объект `Position` было невозможно без предоставления значений `x` и `y`.

Листинг 3.2. Класс Position теперь запрашивает значения x и y в качестве аргументов конструктора

```
final class Position
{
    private int x;
    private int y;

    public function __construct(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public function distanceTo(Position other): float
    {
        return sqrt(
            (other.x - this.x) ** 2 +
            (other.y - this.y) ** 2
        );
    }
}

position = new Position(45, 60);
```

Чтобы получить экземпляр Position, необходимо предоставлять значения x и y

Такой пример реализации конструктора можно использовать для сохранения *инвариантности предметной области* — это значит, что нечто должно быть истинно для заданного объекта, и это нечто определяется знаниями о предметной области и о концепции, которую представляет объект. Инвариантность в рассмотренном примере состоит в том, что позиция задается обеими координатами: x и y.

УПРАЖНЕНИЕ

1 Что не так с объектом Money, показанным ниже?

```
money = new Money()
money.setAmount(100);
money.setCurrency('USD');
```

а В нем используются сеттеры для предоставления минимально необходимых данных.

б В нем нет зависимостей.

в Видимо, у его конструктора есть аргументы со значениями по умолчанию.

г Он может существовать в незавершенном состоянии.

3.2. ЗАПРАШИВАЙТЕ ТОЛЬКО ДАННЫЕ, КОТОРЫЕ ИМЕЮТ СМЫСЛ

В примере выше конструктор принимает любое целое число, положительное или отрицательное, вплоть до бесконечности в обоих направлениях. А теперь рассмотрим другую систему координат, где расположение задается широтой и долготой, определяющими положение объекта на земле. В таком случае не каждое значение для широты и долготы будет считаться осмысленным, см. листинг 3.3.

Листинг 3.3. Класс Coordinates

```
final class Coordinates
{
    private float latitude;
    private float longitude;

    public function __construct(float latitude, float longitude)
    {
        this.latitude = latitude;
        this.longitude = longitude;
    }

    // ...
}

meaningfulCoordinates = new Coordinates(45.0, -60.0);

offThePlanet = new Coordinates(1000.0, -20000.0);
```

Ничто не запрещает создать объект Coordinates, который не будет иметь никакого смысла

Всегда следите, чтобы клиент предоставлял только осмысленные значения. Все, что считается бессмысленным, теоретически можно также назвать инвариантом предметной области. Но в данном случае инвариантом будет то, что «широта — это значение между –90 и 90 включительно, а долгота — значение между –180 и 180 включительно».

При проектировании объектов ориентируйтесь на инварианты предметной области. Собирайте больше информации об инвариантах и используйте их при создании модульных тестов. Например, ниже представлен листинг, в котором используется утилита `expectException()`, описанная в разделе 1.10.

Листинг 3.4. Проверка инвариантов предметной области в классе Coordinates

Ключевое слово, которое должно быть в сообщении исключения

```
expectException(
    InvalidArgumentException.className,
    'Latitude',
```

Тип ожидаемого исключения

```
    function() {
        new Coordinates(90.1, 0.0);
    }
);
expectException(
    InvalidArgumentException.className,
    'Longitude',
    function() {
        new Coordinates(0.0, 180.1);
    }
);
// и т. д...
```

Анонимная функция, которая должна спровоцировать выдачу исключения

Чтобы тесты проходили успешно, выдавайте исключение в конструкторе, как только любой аргумент будет неверным, см. листинг 3.5.

Листинг 3.5. Выдача исключений при недопустимых аргументах конструктора

```
final class Coordinates
{
    // ...

    public function __construct(float latitude, float longitude)
    {
        if (latitude > 90 || latitude < -90) {
            throw new InvalidArgumentException(
                'Широта должна быть в пределах от -90 до 90'
            );
        }
        this.latitude = latitude;

        if (longitude > 180 || longitude < -180) {
            throw new InvalidArgumentException(
                'Долгота должна быть в пределах от -180 до 180'
            );
        }
        this.longitude = longitude;
    }
}
```

Хотя порядок выражений в конструкторе неважен (как мы обсудили выше), рекомендуется все же проводить проверку аргументов непосредственно перед присвоением значения свойству. Это облегчит чтение и понимание того, как связаны выражения.

В некоторых случаях проверки допустимости аргументов по отдельности недостаточно и нужно проверить, что все вместе взятые аргументы конструктора имеют смысл. В примере ниже показан класс `ReservationRequest`, который используется, чтобы хранить информацию о бронировании номеров в отеле.

Листинг 3.6. Класс ReservationRequest

```
final class ReservationRequest
{
    public function __construct(
        int numberOfRooms,
        int numberOfAdults,
        int numberOfChildren
    ) {
        // ...
    }
}
```

Обсуждая бизнес-правила для этого объекта с отраслевым экспертом, вы узнаете следующие ограничения:

- Всегда должен быть хотя бы один взрослый (так как дети не могут бронировать номера в отеле самостоятельно).
- Любой может забронировать для себя отдельный номер, но количество бронирований не должно превышать количества гостей (нет смысла бронировать номер, в котором никто не будет ночевать).

Поэтому получается, что `numberOfRooms` (число номеров) и `numberOfAdults` (число взрослых) связаны и должны проверяться на допустимость вместе. Нужно убедиться, что конструктор принимает оба значения и применяет соответствующие правила, как в листинге ниже.

Листинг 3.7. Проверка аргументов конструктора на допустимость

```
final class ReservationRequest
{
    public function __construct(
        int numberOfRooms,
        int numberOfAdults,
        int numberOfChildren
    ) {
        if (numberOfRooms > numberOfAdults + numberOfChildren) {
            throw new InvalidArgumentException(
                'Количество номеров не должно превышать количества гостей'
            );
        }

        if (numberOfAdults < 1) {
            throw new InvalidArgumentException(
                'numberOfAdults должен иметь значение 1 или больше'
            );
        }

        if (numberOfChildren < 0) {
```

```
            throw new InvalidArgumentException(
                'numberOfChildren должен иметь значение 0 или больше'
            );
        }
    }
}
```

В другом случае аргументы конструктора могут поначалу показаться связанными, но если класс перепроектировать, то консолидированной проверки аргументов удастся избежать. Рассмотрим следующий класс, представляющий бизнес-сделку, в ходе которой необходимо разделить некоторое количество денежных средств между двумя сторонами.

Листинг 3.8. Класс Deal

```
final class Deal
{
    public function __construct(
        int totalAmount,
        int amountToFirstParty,
        int amountToSecondParty
    ) {
        // ...
    }
}
```

Необходимо хотя бы по отдельности проверить аргументы конструктора (например, общее количество (`totalAmount`) должно быть больше 0 и т. д.). Но здесь также присутствует инвариант, который затрагивает все аргументы: сумма средств, получаемых каждой из сторон, должна быть равна общему их количеству. В листинге ниже показано, как проверить это правило.

Листинг 3.9. Deal проверяет сумму средств сторон

```
final class Deal
{
    public function __construct(
        int totalAmount,
        int amountToFirstParty,
        int amountToSecondParty
    ) {
        // ...

        if (amountToFirstParty + amountToSecondParty
            != totalAmount) {
            throw new InvalidArgumentException(/* ... */);
        }
    }
}
```

Как вы, возможно, заметили, это правило можно реализовать намного эффективнее. Общую сумму саму по себе можно и не сообщать, если клиент предоставляет

положительные числа для `amountToFirstParty` и `amountToSecondParty`. Объект `Deal` может сам узнать общую сумму сделки, складывая эти значения. Консолидированной проверки аргументов конструктора не требуется.

Листинг 3.10. Удаление лишних аргументов конструктора

```
final class Deal
{
    private int amountToFirstParty;
    private int amountToSecondParty;

    public function __construct(
        int amountToFirstParty,
        int amountToSecondParty
    ) {
        if (amountToFirstParty <= 0) {
            throw new InvalidArgumentException(/* ... */);
        }
        this.amountToFirstParty = amountToFirstParty;

        if (amountToSecondParty <= 0) {
            throw new InvalidArgumentException(/* ... */);
        }
        this.amountToSecondParty = amountToSecondParty;
    }

    public function totalAmount(): int
    {
        return this.amountToFirstParty
            + this.amountToSecondParty;
    }
}
```

Другой пример, в котором может показаться, что аргументы конструктора необходимо проверять вместе, — класс `Line` в листинге ниже, представляющий линию.

Листинг 3.11. Класс Line

```
final class Line
{
    public function __construct(
        bool isDotted,
        int distanceBetweenDots
    ) {
        if (isDotted && distanceBetweenDots <= 0) {
            throw new InvalidArgumentException(
                'Расстояние между точками должно быть положительным числом'
            );
        }

        // ...
    }
}
```

Нам важно знать расстояние между точками, только если линия является пунктирной. Для сплошных линий расстояния между точками не существует

Реализация будет эффективнее, если предоставить клиенту возможность создавать линии двух видов: пунктирные и сплошные. Различные типы линий можно создавать, вызывая различные конструкторы.

Листинг 3.12. Line теперь предлагает разные способы создания линий

```
final class Line
{
    private bool isDotted;
    private int distanceBetweenDots;

    public static function dotted(int distanceBetweenDots): Line
    {
        if (distanceBetweenDots <= 0) {
            throw new InvalidArgumentException(
                'Расстояние между точками должно быть положительным числом'
            );
        }

        line = new Line(/* ... */);
        line.distanceBetweenDots = distanceBetweenDots;
        line.isDotted = true;

        return line;
    }

    public static function solid(): Line
    {
        line = new Line();

        line.isDotted = false; ◄── Здесь не нужно беспокоиться о distanceBetweenDots

        return line;
    }
}
```

Эти статические методы называют *именованными конструкторами*, и мы рассмотрим их подробнее в разделе 3.9.

УПРАЖНЕНИЕ

2 PriceRange представляет минимальную и максимальную цены в центах, которые покупатель может заплатить за некий объект:

```
final class PriceRange
{
    public function __construct(int minimumPrice, int maximumPrice)
    {
        this.minimumPrice = minimumPrice;
        this.maximumPrice = maximumPrice;
    }
}
```

Конструктор здесь принимает любое значение `int` для обоих аргументов. Улучшите конструктор и сделайте так, чтобы он выдавал ошибку, если эти значения не имеют смысла.

Если предоставить каждому объекту минимально необходимые верные и имеющие смысл данные во время вызова конструктора, приложение будет содержать только полные и действительные объекты, поведение которых будет соответствовать ожиданиям. Никаких сюрпризов или дополнительных проверок.

3.3. НЕ ИСПОЛЬЗУЙТЕ СОБСТВЕННЫЕ КЛАССЫ ИСКЛЮЧЕНИЙ ПРИ ПРОВЕРКЕ НЕДОПУСТИМЫХ АРГУМЕНТОВ

До сих пор мы выдавали общее исключение `InvalidArgumentException`, если аргумент метода не соответствовал нашим ожиданиям. Можно использовать собственный класс исключений, который расширяется от `InvalidArgumentException`. Преимущество такого подхода в том, что можно получать специфичные типы исключений и обрабатывать их определенным образом.

Листинг 3.13. SpecificEsception можно получать и обрабатывать

```
final class SpecificException extends InvalidArgumentException
{
}

try {
    // попытка создать объект
} catch (SpecificException exception) {
    // обработка специфичной задачи определенным образом
}
```

Тем не менее при проверке аргументов это обычно не требуется. Недопустимый аргумент означает, что клиент использует объект не по назначению. Обычно это вызвано ошибкой программирования. В таком случае лучше остановить работу программы, не пытаясь восстановиться, и исправить ошибку.

В то же время в случае с `RuntimeExceptions` зачастую имеет смысл использовать собственные исключения, так как после них есть возможность восстановиться или преобразовать их в сообщения об ошибках, понятные пользователю. Мы подробнее рассмотрим исключения времени исполнения и то, как их создавать, в разделе 5.2.

3.4. ПРОВЕРЯЙТЕ СПЕЦИФИЧЕСКИЕ ИСКЛЮЧЕНИЯ ДЛЯ НЕДОПУСТИМЫХ АРГУМЕНТОВ, АНАЛИЗИРУЯ СООБЩЕНИЯ ИСКЛЮЧЕНИЙ

Даже если вы используете общий класс исключений `InvaliArgumentException` для проверки аргументов методов, эти аргументы необходимо различать во время выполнения модульных тестов. Еще раз посмотрим на конструктор класса `Coordinates`.

Листинг 3.14. Класс Coordinates

```
final class Coordinates
{
    // ...

    public function __construct(float latitude, float longitude)
    {
        if (latitude > 90 || latitude < -90) {
            throw new InvalidArgumentException(
                'Широта должна иметь значение от -90 до 90'
            );
        }
        this.latitude = latitude;

        if (longitude > 180 || longitude < -180) {
            throw new InvalidArgumentException(
                'Долгота должна иметь значение от -180 до 180'
            );
        }
        this.longitude = longitude;
    }
}
```

Нам нужно проверить, что клиенты не могут передавать неверные аргументы. Для этого напишем несколько тестов, как показано ниже.

Листинг 3.15. Тестирование инвариантов предметной области в классе Coordinates

```
// Широта не может быть больше 90.0
expectException(
    InvalidArgumentException.className,
    function() {
        new Coordinates(90.1, 0.0);
    }
);

// Широта не может быть меньше -90.0
expectException(
    InvalidArgumentException.className,
    function() {
        new Coordinates(-90.1, 0.0);
    }
```

```
);

// Долгота не может быть больше 180.0
expectException(
    InvalidArgumentException.className,
    function() {
        new Coordinates(-90.1, 180.1);
    }
);
```

В последнем тесте из конструктора выдается исключение `InvalidArgumentException`, но это не то, что мы ожидали. Так как в этом случае используется недопустимое значение для широты из предыдущего теста, при попытке создания объекта `Coordinates` будет выдаваться исключение, которое сообщит, что «широта должна иметь значение от –90.0 до 90.0». Но тест должен проверять, что код отклонит недопустимые значения долготы. Это значит, что долгота не будет проверяться в этом тестовом сценарии, даже если все тесты пройдут успешно.

Чтобы предотвратить такого рода ошибки, старайтесь проверять, что выдаваемые в модульном тесте исключения соответствуют ожидаемым. Удобнее всего проверять, что сообщение об исключении содержит определенные слова.

Листинг 3.16. Проверка того, что сообщение исключения содержит заданную строку

```
expectException(
    InvalidArgumentException.className,
    'Longitude',
    function() {
        new Coordinates(-90.1, 180.1);
    }
);
```

Это слово (longitude — долгота) должно быть в сообщении об ошибке

При добавлении в тест ожидания подобного сообщения об исключении, как в листинге 3.15, тест будет выдавать ошибку. И он снова будет проходить успешно, если в конструктор будет передано верное значение широты.

3.5. СОЗДАВАЙТЕ НОВЫЕ ОБЪЕКТЫ, ЧТОБЫ ИЗБЕЖАТЬ МНОГОКРАТНОЙ ПРОВЕРКИ ИНВАРИАНТОВ ПРЕДМЕТНОЙ ОБЛАСТИ

На практике вы обнаружите, что одна и та же логика проверки повторяется в одном классе или даже в разных классах. Например, взгляните на следующий класс `User`, в котором при помощи функции из стандартной библиотеки многократно проверяется email-адрес.

Листинг 3.17. Класс User

```
final class User
{
    private string emailAddress;

    public function __construct(string emailAddress)
    {
        if (!is_valid_email_address(emailAddress)) {
            throw new InvalidArgumentException(
                'Invalid email address'
            );
        }
        this.emailAddress = emailAddress;
    }

    // ...

    public function changeEmailAddress(string emailAddress): void
    {
        if (!is_valid_email_address(emailAddress)) {
            throw new InvalidArgumentException(
                'Invalid email address'
            );
        }
        this.emailAddress = emailAddress;
    }
}

expectException(
    InvalidArgumentException.className,
    'email',
    function () {
        new User('not-a-valid-email-address');
    }
);

user = new User('valid@emailaddress.com');

expectException(
    InvalidArgumentException.className,
    'email',
    function () use (user) {
        user.changeEmailAddress('not-a-valid-email-address');
    }
);
```

Проверка, что предоставленный email-адрес является допустимым

Повторная проверка, если адрес будет меняться

Конструктор будет выдавать исключения для недопустимых адресов

Сначала создается допустимый объект User

Метод changeEmailAddress() также будет выдавать исключение при недопустимом email-адресе

И хотя можно просто передать логику проверки email-адреса в отдельный метод, более грамотное решение — создать отдельный класс объектов, представляющий собой допустимый email-адрес. Так как мы ожидаем, что все объекты создаются допустимыми, уберем часть «valid» из имени класса и реализуем решение следующим образом.

Листинг 3.18. Класс EmailAddress

```
final class EmailAddress
{
    private string emailAddress;

    public function __construct(string emailAddress)
    {
        if (!is_valid_email_address(emailAddress)) {
            throw new InvalidArgumentException(
                'Недопустимый email'
            );
        }
        this.emailAddress = emailAddress;
    }
}
```

Когда вам встретится объект `EmailAddress`, вы всегда будете знать, что значение email-адреса для него уже проверено:

```
final class User
{
    private EmailAddress emailAddress;

    public function __construct(EmailAddress emailAddress)
    {
        this.emailAddress = emailAddress;
    }

    // ...

    public function changeEmailAddress(EmailAddress emailAddress): void
    {
        this.emailAddress = emailAddress;
    }
}
```

Оборачивание значений в новые объекты под названием *объекты-значения* полезно не только, чтобы избегать повторения логических конструкций. Как только вы заметите, что метод принимает примитивные значения (`string`, `int` и т. д.), стоит задуматься о том, чтобы создать для них класс. Чтобы принять решение, ответьте на вопрос: будут ли здесь приемлемы значения `string`, `int`, и т. д.? Если ответ — нет, создавайте отдельный класс.

Объекты-значения сами по себе стоит рассматривать как типы наравне со `string`, `int` и т. п. Создавая новые объекты для представления понятий, вы расширяете систему типов. Компилятор языка или среда исполнения могут проводить проверку соответствия типов, чтобы только правильные типы использовались при передаче значений через аргументы методов и в возвращаемых значениях.

УПРАЖНЕНИЕ

3 Код страны может быть представлен строкой из двух символов, но не каждая такая строка может быть кодом страны. Создайте класс объекта-значения, который будет представлять код страны. Будем считать, что список известных кодов страны состоит из `NL` и `GB`.

3.6. СОЗДАВАЙТЕ НОВЫЕ ОБЪЕКТЫ ДЛЯ ПРЕДСТАВЛЕНИЯ СОСТАВНЫХ ЗНАЧЕНИЙ

При создании новых типов вы обнаружите, что некоторые из них связаны друг с другом и всегда передаются вместе во всех методах. Например, количество денег всегда сопровождается сведениями о валюте, как в листинге ниже. Если в метод будет передано лишь количество, он не будет иметь представления, как это количество обрабатывать.

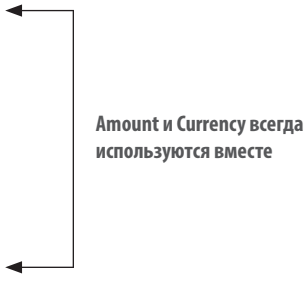
Листинг 3.19. Amount и Currency

```
final class Amount
{
    // ...
}

final class Currency
{
    // ...
}

final class Product
{
    public function setPrice(
        Amount amount,
        Currency currency
    ): void {
        // ...
    }
}

final class Converter
{
    public function convert(
        Amount localAmount,
        Currency localCurrency,
        Currency targetCurrency
    ): Amount {
        // ...
    }
}
```



В последнем примере возвращаемый тип неочевиден. Возвращаться здесь будет `Amount`, а что касается валюты, то ожидается, что она будет соответствовать заданному `targetCurrency`. Но это не очевидно, если судить лишь по использованию типов в методе.

Если вы видите, что значения взаимозависимые (или всегда используются вместе), оберните их в один тип. В случае с `Amount` и `Currency` подходящим названием для этой комбинации может быть «money», что мы и передадим в названии класса.

Листинг 3.20. Класс Money

```
final class Money
{
    public function __construct(Amount amount, Currency currency)
    {
        // ...
    }
}
```

Данный тип явно сообщает, что эти значения хранятся вместе, хотя при желании их можно использовать отдельно.

ПРИ ДОБАВЛЕНИИ НОВЫХ ТИПОВ ОБЪЕКТОВ ПРИДЕТСЯ ВВОДИТЬ БОЛЬШЕ КОДА. ЕСТЬ ЛИ В ЭТОМ НЕОБХОДИМОСТЬ?

В строке `new Amount(100)` больше символов, чем в строке `100`, но, напечатав их, мы сможем использовать преимущества объектов:

1. Данные, хранящиеся в объекте, обязательно будут проверены.
2. Объект обычно предоставляет дополнительные, осмысленные варианты поведения, использующие его данные.
3. Объект может хранить совместно используемые значения.
4. Объект помогает скрывать детали реализации от клиентов.

Если все эти хлопоты по созданию объектов для примитивных значений кажутся вам лишними, вы всегда можете задать вспомогательные методы. Например:

```
// Было:

money = new Money(new Amount(100), new Currency('USD'));

// Стало:

money = Money.create(100, 'USD');
```

Подробнее о таком стиле создания объектов вы узнаете в разделе 3.9.

УПРАЖНЕНИЕ

4 При помощи объекта Run можно сохранять преодоленное на пробежке расстояние:

```
final class Run
{
    public function __construct(int distance)
    {
        // ...
    }
}
```

Но в данной реализации есть проблема: невозможно узнать, какого рода значение представляет аргумент `distance`. Измеряется ли он в метрах, футах или, может, километрах? Для решения нужен объект-значение, который будет представлять как количество, так и единицу измерения преодоленного расстояния. В рамках нашей реализации примем, что расстояние может измеряться в метрах или футах.

3.7. ИСПОЛЬЗУЙТЕ ПРОВЕРКИ УТВЕРЖДЕНИЙ ДЛЯ АРГУМЕНТОВ КОНСТРУКТОРА

Мы уже встречались с примерами конструкторов, в которых выдаются исключения, если что-то идет не так. Общая структура выглядит следующим образом:

```
if (somethingIsWrong()) {
    throw new InvalidArgumentException(/* ... */);
}
```

Эти проверки в самом начале метода называются проверками утверждений (assertion) и представляют собой базовые проверки безопасности. Их можно использовать, чтобы понять ситуацию, изучить материалы и сообщить, что именно пошло не так. Поэтому их также называют проверкой предусловий. После успешных проверок предусловий можно смело выполнять задачи со всеми предоставленными данными.

Так как вы будете выполнять одни и те же проверки в разных местах кода, удобно использовать библиотеку проверок[1]. В такой библиотеке содержится множество

[1] Если вы программируете на PHP, воспользуйтесь пакетом `beberlei/assert` или `webmozart/assert`.

универсальных выражений для большинства практических ситуаций. Вот несколько примеров:

```
Assertion.greaterThan(value, limit);
Assertion.isCallable(value);
Assertion.between(
    value,
    lowerLimit,
    upperLimit
);
// и т. д. ...
```

Нужно ли вводить все эти проверки в модульные тесты объекта? Исходите из ответа на вопрос: может ли в теории среда исполнения распознать эту проблему? Если ответ — да, то модульный тест не нужен.

Например, в динамически типизированных языках, таких как PHP, нет возможности указать тип аргумента для `list of <classname>`. Вместо этого используется довольно общий тип `array`. Чтобы убедиться, что массив действительно является простым списком объектов определенного типа, следует применить проверку утверждений, как в листинге ниже.

Листинг 3.21. EventDispatcher использует функцию проверки утверждений в конструкторе

```
final class EventDispatcher
{
    public function __construct(array eventListeners)
    {
        Assertion.allIsInstanceOf(
            eventListeners,
            EventListener.className
        );

        // ...
    }
}
```

Так как это состояние ошибки, которое способна уловить продвинутая система типов, не нужно писать модульный тест, выявляющий исключение `AssertionFailedException`, выдаваемое методом `allIsInstanceOf()`. Но если нужно проверить, что заданное значение находится в определенном диапазоне, или проверить количество сущностей в списке и т. д., то придется создать модульные тесты, чтобы убедиться, что учтены граничные случаи. Можно заметить, что в предыдущем примере необходимо учитывать в проверке инвариант предметной области, когда заданная широта должна быть в промежутке между –90 и 90 включительно.

Листинг 3.22. Добавление модульного теста для проверки инварианта предметной области

```
expectException(
    AssertionFailedException.className,
    'latitude',
    function() {
        new Coordinates(-90.1, 0.0)
    }
);
// и т. д. ...
```

НЕ СОБИРАЙТЕ ИСКЛЮЧЕНИЯ

Не стоит собирать исключения, а затем выдавать их в виде списка, хотя некоторые инструменты и позволяют это делать. Проверка не означает, что в результате пользователь получит наглядный список ошибок. Она предназначена для программиста, которому нужно знать, что конструктор или метод используется неправильно. Как только вы заметите что-то странное, просто сделайте так, чтобы объект кричал об этом.

Если вы хотите, чтобы пользователи узнали, что не так с данными, которые они предоставили (при отправке формы, API-запроса и т. п.), стоит использовать объекты для передачи данных (data transfer object, DTO). Мы обсудим этот тип объектов в конце главы.

УПРАЖНЕНИЕ

5 Доступны следующие функции проверки утверждений:

```
Assertion.greaterThan(value, limit);
Assertion.between(
    value,
    lowerLimit,
    upperLimit
);
Assertion.lessThan(value, limit);
```

Перепишите конструктор `PriceRange` с использованием функции проверки утверждений.

```
final class PriceRange
{
    public function __construct(int minimumPrice, int maximumPrice)
    {
        if (minimumPrice < 0) {
            throw new InvalidArgumentException(
                'minimumPrice должен иметь значение 0 или больше'
            );
```

```
            }
            if (maximumPrice < 0) {
                throw new InvalidArgumentException(
                    'maximumPrice должен иметь значение 0 или больше'
                );
            }
            if (maximumPrice <= minimumPrice) {
                throw new InvalidArgumentException(
                    'maximumPrice должен быть больше, чем minimumPrice'
                );
            }

            this.minimumPrice = miminumPrice;
            this.maximumPrice = maximumPrice;
        }
    }
```

3.8. НЕ ВНЕДРЯЙТЕ ЗАВИСИМОСТИ, А ПЕРЕДАВАЙТЕ ИХ В КАЧЕСТВЕ АРГУМЕНТОВ МЕТОДОВ

У сервисов могут быть зависимости, и их нужно внедрять в качестве аргументов конструктора. Но другим типам объектов вместо зависимостей следует передавать значения, объекты-значения и их списки. Если объекту-значению все еще нужен сервис для выполнения задачи, можно внедрить его в качестве аргумента метода, как в листинге ниже.

Листинг 3.23. Классу Money нужен сервис ExchangeRateProvider

```
final class Money
{
    private Amount amount;
    private Currency currency;

    public function __construct(Amount amount, Currency currency)
    {
        this.amount = amount;
        this.currency = currency;
    }

    public function convert(
        ExchangeRateProvider exchangeRateProvider,
        Currency targetCurrency
    ): Money {
        exchangeRate = exchangeRateProvider.getRateFor(
             this.currency,
            targetCurrency
        );

        return exchangeRate.convert(this.amount);
    }
}
```

ExchangeRateProvider — аргумент метода, а не конструктора

Может показаться странным передавать сервис через аргументы метода, поэтому можно поступить иначе: передавать не сервис `ExchangeRateProvider`, а лишь информацию, которую мы от него получаем: `ExchangeRate`. Для этого потребуется класс `Money`, который будет представлять объекты `Amount` и `Currency`, но это разумная плата за то, чтобы не внедрять зависимости. Примерный результат см. ниже.

Листинг 3.24. Альтернативная реализация: без передачи сервиса ExchangrRateProvider

```
final class ExchangeRate
{
    public function __construct(
        Currency from,
        Currency to,
        Rate rate
    ) {
        // ...
    }

    public function convert(Amount amount): Money
    {
        // ...
    }
}

money = new Money(/* ... */);
exchangeRate = exchangeRateProvider.getRateFor(
    money.currency(),
    targetCurrency
);
converted = exchangeRate.convert(money.amount());
```

Получаем ExchangerRate

Затем используем его для преобразования имеющегося количества

После всех этих изменений остановимся на решении, которое предполагает использование лишь внутреннего объекта `Currency` класса `Money` (а не `Amount`), как это сделано в листинге ниже. (Мы вернемся к теме передачи внутренних сущностей объекта в разделе 6.3.)

Листинг 3.25. Передача ExchangeRate вместо ExchangeRateProvider

```
final class Money
{
    public function convert(ExchangeRate exchangeRate): Money
    {
        Assertion.equals(
            this.currency,
            exchangeRate.fromCurrency()
        );

        return new Money(
            exchangeRate.rate().applyTo(this.amount),
            exchangeRate.targetCurrency()
```

```
        );
    }
}

money = new Money(/* ... */);
exchangeRate = exchangeRateProvider.getRateFor(
    money.currency(),
    targetCurrency
);
converted = money.convert(exchangeRate);
```

Можно возразить, что в этом решении сведения о деньгах и обменном курсе выражены более четко. Например, конвертированной сумме будет присвоена целевая валюта, а исходной валютой будет валюта исходной суммы.

В некоторых случаях необходимость передачи сервисов в качестве аргументов методов может подтолкнуть к мысли о том, что вместо внедрения сервисов стоит реализовать нужное поведение. Для конвертирования денежных единиц в некую валюту можно создать сервис и передать ему всю работу по сбору необходимой информации из предоставляемых объектов `Amount` и `Currency`.

Листинг 3.26. Альтернативная реализация: всю работу осуществляет ExchangeService

```
final class ExchangeService
{
    private ExchangeRateProvider exchangeRateProvider;

    public function __construct(
        ExchangeRateProvider exchangeRateProvider
    ) {
        this.exchangeRateProvider = exchangeRateProvider;
    }

    public function convert(
        Money money,
        Currency targetCurrency
    ): Money {
        exchangeRate = this.exchangeRateProvider
            .getRateFor(money.currency(), targetCurrency);

        return new Money(
            exchangeRate.rate().applyTo(money.amount()),
            targetCurrency
        );
    }
}
```

Какое решение подойдет именно вам, зависит от того, насколько близко друг к другу вы желаете хранить поведение объектов и данные, хотите ли вы, чтобы такой объект, как `Money`, также имел сведения об обменном курсе, или стремитесь избежать раскрытия внутренних данных объектов.

УПРАЖНЕНИЕ

6 Имея следующий класс `User`, как вы передадите ему сервис `PasswordHasher`?

```
interface PasswordHasher
{
    public function hash(string password): string
}

final class User
{
    private string username;
    private string hashedPassword;

    public function __construct(string username)
    {
        this.username = username;
    }

    public function setPassword(
        string plainTextPassword
    ): void {
        this.hashedPassword = /* ... */;
    }
}
```

Здесь хотелось бы использовать сервис PasswordHasher для хеширования пароля

а Добавив еще один аргумент конструктора:

```
private PasswordHasher hasher;

public function __construct(
    string username,
    PasswordHasher hasher
) {
    this.hasher = hasher;
}

public function setPassword(
    string plainTextPassword
): void {
    this.hashedPassword = this.hasher.hash(
        plainTextPassword
    );
}
```

б Добавив метод `setPasswordHasher` (`PasswordHasher passwordHasher`):

```
private PasswordHasher hasher;

public function setPasswordHasher(PasswordHasher hasher): void
```

```
{
    this.hasher = hasher;
}

public function setPassword(
    string plainTextPassword
): void {
    this.hashedPassword = this.hasher.hash(
        plainTextPassword
    );
}
```

в Передавая `PasswordHasher` в качестве аргумента метода:

```
public function setPassword(
    string plainTextPassword,
    PasswordHasher hasher
): void {
    this.hashedPassword = hasher.hash(
        plainTextPassword
    );
}
```

г Делая `PasswordHasher` глобально доступным:

```
public function setPassword(
    string plainTextPassword
): void {
    this.hashedPassword = PasswordHasher.getInstance()
        .hash(
            plainTextPassword
        );
}
```

3.9. ИСПОЛЬЗУЙТЕ ИМЕНОВАННЫЕ КОНСТРУКТОРЫ

Для сервисов вполне нормально использовать стандартный способ объявления конструкторов (`public function __construct()`). Тем не менее для других типов объектов рекомендуется использовать *именованные конструкторы*. Они представляют собой публичные статические методы, которые возвращают экземпляр. Их можно считать фабриками объектов.

3.9.1. Создавайте объекты из значений примитивного типа

Обычно именованные конструкторы используются для создания объектов из одного и более значений примитивного типа. В результате получаем такие методы, как `fromString()`, `fromInt()`. Например, взгляните на следующий класс `Date`.

Листинг 3.27. Класс Date оборачивает строку с датой

```
final class Date
{
    private const string FORMAT = 'd/m/Y';
    private DateTime date;

    private function __construct()
    {
        // ничего не делать
    }

    public static function fromString(string date): Date
    {
        object = new Date();

        DateTime = DateTime.createFromFormat(
            Date.FORMAT,
            date
        );

        object.date = DateTime;

        return object;
    }
}

date = Date.fromString('1/4/2019');
```

Нам все равно придется проверить, что createFromFormat() не возвращает false

Важно также добавлять стандартный, но приватный метод-конструктор, чтобы клиенты не могли обойти предложенный именованный конструктор, который может оставить объект в недопустимом или неполном состоянии.

ПОДОЖДИТЕ, А ЭТО РАЗВЕ РАБОТАЕТ?

Может показаться странным, что метод `public static toString()` может создать новый экземпляр объекта и управлять свойством `date` нового экземпляра. Свойство объявлено с модификатором `private`, и эти операции не должны быть разрешены, так ведь?

Инкапсуляция методов и свойств обычно основана на классах, а не на экземплярах, поэтому приватные свойства могут использоваться любым объектом, при условии что эти объекты относятся к одному и тому же классу. Метод `fromString()` в этом примере считается методом того же класса, именно поэтому он может непосредственно использовать свойство `date` без необходимости делать это через сеттер.

3.9.2. Не добавляйте без необходимости такие методы, как toString() или toInt()

Когда вы добавляете именованный конструктор, создающий объект из значений примитивного типа, в целях симметрии вы, возможно, захотите добавить метод, который преобразует объект обратно в примитивные значения. Например, имея метод `fromString(),` можно добавить метод `toString().` Но убедитесь сначала, что это действительно нужно.

3.9.3. Продумывайте и внедряйте понятия, специфичные для предметной области

Когда вы будете обсуждать с отраслевым экспертом формирование клиентских заказов, он никогда не станет говорить о «проектировании» заказа. Возможно, он будет говорить о «создании» заказа или использует более специфичный термин «размещение» заказа (placing a sales order). Обращайте внимание на эти слова и используйте их в названии методов именованных конструкторов.

Листинг 3.28. В реальной жизни заказы клиентов не «проектируются» (construct), а «размещаются» (place)

```
final class SalesOrder
{
    public static function place(/* ... */): SalesOrder
    {
        // ...
    }
}

salesOrder = SalesOrder.place(/* ... */);
```

3.9.4. Для введения ограничения можно использовать приватный конструктор

Для создания некоторых объектов можно использовать разные конструкторы, поскольку создавать такие объекты можно разными способами. Например, если вам нужно округленное десятичное значение, вы можете выбрать целочисленный тип с округлением в большую сторону в качестве стандартного способа представления такого числа. В то же время вы можете разрешить клиентам использовать их текущие значения — строки или числа с плавающей точкой — и далее уже работать с ними как с десятичными. Приватный конструктор помогает убедиться, что объект будет полным и согласованным независимо от метода его создания. Ниже представлен пример.

Листинг 3.29. Включение инвариантов предметной области в приватный конструктор

```
final class DecimalValue
{
    private int value;
    private int precision;

    private function __construct(int value, int precision)
    {
        this.value = value;

        Assertion.greaterOrEqualThan(precision, 0);
        this.precision = precision;
    }

    public static function fromInt(
        int value,
        int precision
    ): DecimalValue {
        return new DecimalValue(value, precision);
    }

    public static function fromFloat(
        float value,
        int precision
    ): DecimalValue {
        return new DecimalValue(
            (int)round(value * pow(10, precision)),
            precision
        );
    }

    public static function fromString(string value): DecimalValue
    {
        result = preg_match('/^(\d+)\.(\d+)/', value, matches);
        if (result == 0) {
            throw new InvalidArgumentException(/* ... */);
        }

        wholeNumber = matches[1];
        decimals = matches[2];

        valueWithoutDecimalSign = wholeNumber . decimals;

        return new DecimalValue(
            (int)valueWithoutDecimalSign,
            strlen(decimals)
        );
    }
}
```

Итак, использование именованных конструкторов дает два главных преимущества:

- Их можно использовать для предоставления нескольких способов создания объекта.
- Кроме того, их можно использовать для включения понятий предметной области в создание объектов.

Кроме создания сущностей и объектов-значений, именованные конструкторы можно применять для введения нескольких типов инициализации собственных исключений. Мы обсудим это позже, в разделе 5.2.

УПРАЖНЕНИЕ

7 Класс `Date`, представленный ниже, можно инстанцировать с передачей строки в правильном формате, которая затем будет преобразована в экземпляр `DateTime`. Но что, если у клиента уже есть объект `DateTime`? Как позволить клиенту передавать свой экземпляр непосредственно в объект `Date` вместо того, чтобы передавать дату в строке?

```
final class Date
{
    private DateTime date;

    public function __construct(string date)
    {
        this.date = DateTime.createFromFormat(
            'd/m/Y',
            date
        );
    }
}
```

а Удалить тип `string` из параметра `date` в конструкторе, чтобы позволить клиентам передавать экземпляр `DateTime`, в котором не будет опечаток.

б Добавить в класс два именованных конструктора: `fromString(string date): Date` и `fromDateTime (DateTime dateTime) : Date`.

в Сделать параметр `string date` необязательным и добавить в конструктор второй необязательный параметр `DateTime dateTime`.

г Создать новый класс, который расширяется от `Date` и переопределяет конструктор на прием экземпляра `DateTime` вместо строки.

3.10. НЕ ИСПОЛЬЗУЙТЕ ЗАПОЛНИТЕЛИ СВОЙСТВ

Если применять все правила проектирования, описанные в этой книге, создаваемые объекты будут полностью контролировать данные, которые в них поступают, которые в них остаются, и операции, которые с ними может выполнять клиент. Но есть и другая модель, которая работает абсолютно противоположно такому стилю проектирования, — использование методов заполнения свойств, таких как метод `fromArray()` в листинге ниже.

Листинг 3.30. В классе Position существует заполнитель свойства под названием fromArray()

```
final class Position
{
    private int x;
    private int y;

    public static function fromArray(array data): Position
    {
        position = new Position();
        position.x = data['x'];
        position.y = data['y'];
        return position;
    }
}
```

Такой метод также можно превратить в универсальный инструмент, копирующий значения из массива `data` и передающий их в соответствующие свойства при помощи рефлексии. Хотя это кажется удобным, внутренние элементы объекта в таком случае будут раскрыты; поэтому всегда проверяйте, что создание объекта полностью контролируется самим объектом.

ПРИМЕЧАНИЕ

В конце главы мы рассмотрим исключение из этого правила. В объектах для передачи данных заполнителем может выступать ассоциативный массив, например для передачи данных формы в объект. Для такого объекта не нужно защищать внутренние данные, как это делается у сущностей или объектов-значений.

3.11. ДОБАВЛЯЙТЕ В ОБЪЕКТ ТОЛЬКО ТО, ЧТО НУЖНО

Начиная проектировать объект, мы обычно задумываемся, что в него нужно добавить. Проектируя сервисы, вы рискуете внедрить больше зависимостей, чем требуется, а это значит, что внедрять нужно только строго необходимые зависимости. То же самое касается и других типов объектов: не запраши-

вайте больше данных, чем это строго необходимо для реализации поведения объекта.

Один из типов объектов, в котором зачастую появляется больше данных, чем требуется, — это событийные объекты, представляющие события приложения. Пример такого события показан в классе `ProductCreated`.

Листинг 3.31. Класс ProductCreated представляет собой событие

```
final class ProductCreated
{
    public function __construct(
        ProductId productId,
        Description description,
        StockValuation stockValuation,
        Timestamp createdAt,
        UserId createdBy,
        /* ... */
    ) {
        // ...
    }
}

this.recordThat(                    <-- Внутри сущности Product
    new ProductCreated(
        /* ... */                   <-- Передаются все данные, которые доступны
    )                                   при создании объекта-значения
);
```

Если вы не знаете, какое событие будет важным для реализации прослушивателей событий, то не добавляйте ничего. Просто объявите конструктор без аргументов, а затем, если понадобится, добавьте больше данных при необходимости. Таким образом, вы будете предоставлять данные, которые действительно будут необходимы.

Как узнать, какие данные должны на самом деле передаваться в конструктор объекта? Путем проектирования через тестирование. Это значит, что сначала нужно знать, как объект будет использоваться.

3.12. НЕ ТЕСТИРУЙТЕ КОНСТРУКТОРЫ

Написание тестов для объектов, в которых вы указываете ожидаемое поведение, позволит понять, какие данные будут нужны при вызове конструктора, а какие данные нужно будет предоставить позже. Это также поможет понять, какие данные будут раскрываться, а какие должны остаться скрытыми деталями реализации объекта.

Например, взглянем на уже знакомый класс `Coordinates`.

Листинг 3.32. Конструктор класса Coordinates

```
final class Coordinates
{
    // ...

    public function __construct(float latitude, float longitude)
    {
        if (latitude > 90 || latitude < -90) {
            throw new InvalidArgumentException(
                'Широта должна быть от -90 до 90'
            );
        }
        this.latitude = latitude;

        if (longitude > 180 || longitude < -180) {
            throw new InvalidArgumentException(
                'Долгота должна быть от -180 до 180'
            );
        }
        this.longitude = longitude;
    }
}
```

Как проверить, что конструктор работает? Рассмотрим следующий тест.

Листинг 3.33. Первая попытка тестирования конструктора класса Coordinates

```
public function it_can_be_constructed(): void
{
    coordinates = new Coordinates(60.0, 100.0);

    assertIsInstanceOf(Coordinates.className, coordinates);
}
```

Не очень информативно. И на самом деле проверка утверждений не завершается ошибкой, пока конструктор не выдаст исключение, а это уже не тот процесс, который мы здесь тестируем.

Какова задача конструктора? Судя по коду, передавать значения свойствам в виде аргументов конструктора. Но как убедиться, что он работает? Можно добавить геттеры, которые позволят узнать информацию о свойствах объекта, как показано ниже.

Листинг 3.34. Дополнительные геттеры для тестирования конструктора класса Coordinates

```
final class Coordinates
{
    // ...
```

```
    public function latitude(): float
    {
        return this.latitude;
    }

    public function longitude(): float
    {
        return this.longitude;
    }
}
```

В листинге ниже показано, как использовать эти геттеры в модульном тесте.

Листинг 3.35. Использование геттеров в модульном тесте

```
public function it_can_be_constructed(): void
{
    coordinates = new Coordinates(60.0, 100.0);

    assertEquals(60.0, coordinates.latitude());
    assertEquals(100.0, coordinates.longitude());
}
```

Но теперь мы создали возможность для выхода внутренних данных за пределы самого объекта, и все это только ради тестирования конструктора.

Посмотрим, что мы делали: мы тестировали код конструктора после его написания. Мы тестировали этот код, зная о том, что происходит в самом классе. Это значит, что реализация теста очень близка к реализации самого класса. Мы передавали данные объекту, не зная, пригодятся ли они нам снова. В итоге мы сделали слишком много и слишком быстро без достаточного дистанцирования от реализации самого объекта.

Единственное, что действительно можно и нужно делать, — проверять, что конструктор не принимает недопустимые аргументы. Мы это уже обсуждали: нужно проверить, что предоставленные значения для широты и долготы, которые выходят за рамки допустимых значений, провоцируют выдачу исключения, что делает невозможным создание объекта `Coordinates`.

Скоро мы еще поговорим о раскрытии данных, но пока прислушайтесь к следующим советам:

- Тестируйте конструктор только на те случаи, где определенно должна произойти ошибка.
- Передавайте только те данные в виде аргументов конструктора, которые непосредственно необходимы для реализации поведения объекта.

- Добавляйте дополнительные геттеры, только если нужно раскрывать внутренние данные другому клиенту, а не в целях тестирования.

Когда вы начнете добавлять фактическое поведение объекта, вы в любом случае будете тестировать позитивный сценарий, так как для этого вам будет необходим полностью инстанцированный объект.

УПРАЖНЕНИЕ

8 Что не так с кодом сущности `Product`, представленным ниже?

```
final class Product
{
    private int id;
    private string name;

    public function __construct(int id, string name)
    {
        this.id = id;
        this.name = name;
    }

    public function id(): int
    {
        return this.id;
    }

    public function name(): string
    {
        return this.name;
    }
}

public function it_can_be_constructed(): void
{
    product = new Product(1, 'Some name');

    assertEquals(1, product.id());
    assertEquals('Some name', product.name());
}
```

Это единственный тест для класса Product

а У него есть геттеры.

б Кажется, что геттеры здесь нужны лишь для тестирования конструктора.

в Свойства не могут иметь значение `null`.

3.13. ИСКЛЮЧЕНИЕ ИЗ ПРАВИЛА: ОБЪЕКТЫ ДЛЯ ПЕРЕДАЧИ ДАННЫХ

Правила, описанные в этой главе, применимы к сущностям и объектам-значениям; мы уделяем пристальное внимание тому, насколько последовательные и правильные данные предоставляются объекту. Эти объекты могут гарантировать правильное поведение, только если используемые ими данные тоже корректны.

Существует еще один тип объектов, о котором мы пока не говорили и к которому не применимы правила, описанные ранее. Такой тип объекта вы будете встречать в граничных областях приложения, там, где данные из внешнего мира будут преобразовываться в структуры, с которыми способно работать приложение. Природа таких процессов обусловливает отличие их поведения от поведения сущностей и объектов-значений.

Этот особенный тип объектов называется объектами для передачи данных (Data transfer object, DTO):

- DTO можно создавать при помощи обычного конструктора.
- Его свойства можно заполнять одно за другим.
- Все его свойства раскрыты.
- Его свойства содержат только примитивные значения.
- Свойства могут содержать другие DTO или простые массивы DTO.

3.13.1. Используйте публичный модификатор для свойств

Так как DTO не защищает свое состояние и раскрывает все свои свойства, нет необходимости добавлять геттеры и сеттеры. А это значит, что вполне достаточно просто использовать для свойств модификатор `public`. Так как DTO могут создаваться постепенно и им не нужно предоставлять минимальное количество данных, они не требуют методов-конструкторов.

DTO часто используются в качестве командных объектов, они соответствуют пользовательскому назначению и содержат все необходимые для этого данные. Примером такого командного объекта выступает следующий класс — `ScheduleMeetup`, который представляет собой намерение пользователя назначить митап с заданной темой и указанной датой.

Листинг 3.36. DTO класса ScheduleMeetup

```
final class ScheduleMeetup
{
    public string title;
    public string date;
}
```

Можно использовать такой объект, например, заполняя его данными отправляемых форм, а затем передавая его сервису, который назначит митап для пользователя. Пример такой реализации см. в листинге ниже.

Листинг 3.37. Заполнение DTO класса ScheduleMeetup и его передача сервису

```
final class MeetupController
{
    public function scheduleMeetupAction(Request request): Response
    {
        formData = /* ... */;

        scheduleMeetup = new ScheduleMeetup();
        scheduleMeetup.title = formData['title'];
        scheduleMeetup.date = formData['date'];

        this.scheduleMeetupService.execute(scheduleMeetup);

        // ...
    }
}
```

Извлечение данных формы из тела запроса

Создание командного объекта при помощи этих данных

Этот сервис создаст сущность и несколько объектов значений и будет их хранить. После инстанцирования эти объекты будут выдавать исключения, если что-то пойдет не так с данными, которые им предоставляют. Тем не менее такие исключения не будут слишком дружественны пользователю, их даже нельзя будет легко перевести на язык пользователя. Кроме того, так как они нарушают естественный поток приложения, исключения нельзя собрать и вернуть пользователю в виде списка ошибок ввода.

3.13.2. Не выдавайте исключения, а собирайте ошибки при проверках

Если вы хотите, чтобы пользователи могли исправлять все свои ошибки за один раз, перед отправкой формы, стоит проверить данные команды перед передачей объекта сервису, который будет его обрабатывать. Один из способов это сделать — добавить в объект метод `validate()`, который может возвращать простой список сообщений об ошибках. Если список пуст, то предоставленные значения допустимы.

Листинг 3.38. Проверка DTO класса ScheduleMeetup

```
final class ScheduleMeetup
{
    public string title;
    public string date;

    public function validate(): array
    {
        errors = [];

        if (this.title == '') {
            errors['title'][] = 'validation.empty_title';
        }

        if (this.date == '') {
            errors['date'][] = 'validation.empty_date';
        }

        DateTime.createFromFormat('d/m/Y', this.date);
        errors = DateTime.getLastErrors();
        if (errors['error_count'] > 0) {
            errors['date'][] = 'validation.invalid_date_format';
        }

        return errors;
    }
}
```

Формы и проверочные библиотеки содержат более удобные инструменты для проверок. Например, компоненты Symphony Form и Validator прекрасно работают с объектами для передачи данных.

3.13.3. Если нужно, используйте заполнение свойств

Ранее мы обсуждали заполнители свойств и то, что их не стоит использовать при работе с большинством типов объектов, так как они раскрывают все внутренние данные объектов. В случае с DTO это не проблема, поскольку DTO и так не защищает свое состояние. Поэтому, если нужно, добавьте в DTO методы-заполнители для свойств, например для копирования данных формы или данных JSON-запроса, непосредственно в командный объект. Поскольку заполнение свойств — это первое, что должно произойти в объекте DTO, имеет смысл реализовать заполнитель свойства в виде именованного конструктора.

Листинг 3.39. У DTO класса ScheduleMeetup есть заполнитель свойства

```
final class ScheduleMeetup
{
    public string title;
    public string date;
```

```
    public static function fromFormData(
        array formData
    ): ScheduleMeetup {
        scheduleMeetup = new ScheduleMeetup();

        scheduleMeetup.title = formData['title'];
        scheduleMeetup.date = formData['date'];

        return scheduleMeetup;
    }
}
```

УПРАЖНЕНИЯ

9 Какой нужен тип объекта, если вы хотите передавать пользователю список ошибок?

- **а** Сущность.
- **б** DTO.

10 Какой тип объекта будет выдавать исключение, если предоставленные данные будут некорректными?

- **а** Сущность.
- **б** DTO.

11 Какой тип объекта может ограничить раскрываемые данные?

- **а** Сущность.
- **б** DTO.

ЗАКЛЮЧЕНИЕ

- Объекты, которые не являются сервисами, получают значения или объекты-значения, а не зависимости. При вызове конструктора для согласованного поведения создаваемого объекта требуется минимальное количество данных. Если какие-то из предоставленных аргументов являются недопустимыми, конструктор должен выдавать исключение.
- Оборачивание примитивных значений аргументов в объекты-значения помогает упростить повторное использование правил для проверок таких значений. Кроме того, указание специфического для предметной области имени класса или значения добавляет больше осмысленности в код.
- Конструкторы объектов, которые не являются сервисами, должны быть статическими методами, или *именованными конструкторами*, которые также

предоставляют дополнительную возможность для введения большего количества имен, соответствующих предметной области.

- Не следует передавать конструктору больше данных, чем необходимо для создания объекта, поведение которого соответствует заданному в модульных тестах.
- Большинство указанных правил не относятся к объектам для передачи данных (DTO). DTO используются для передачи данных из внешнего мира, и они раскрывают все свои внутренние данные.

ОТВЕТЫ К УПРАЖНЕНИЯМ

1 Правильные ответы: **a** и **r**. `Money` — это не сервис, поэтому у него не должно быть зависимостей, передаваемых в качестве аргументов конструктора. Также, судя по примеру, невозможно понять, есть ли у конструктора значения аргументов по умолчанию.

2 Вариант ответа:

```
final class PriceRange
{
    public function __construct(int minimumPrice, int maximumPrice)
    {
        if (minimumPrice < 0) {
            throw new InvalidArgumentException(
                'minimumPrice должен быть 0 или больше'
            );
        }
        if (maximumPrice < 0) {
            throw new InvalidArgumentException(
                'maximumPrice должен быть 0 или больше'
            );
        }
        if (minimumPrice > maximumPrice) {
            throw new InvalidArgumentException(
                'maximumPrice должен быть больше, чем minimumPrice'
            );
        }
            this.minimumPrice = miminumPrice;
            this.maximumPrice = maximumPrice;
    }
}
```

3 Вариант ответа:

```
final class CountryCode
{
    private static knownCountryCodes = ['NL', 'GB'];
```

```
    private string countryCode;

    public function __construct(string countryCode)
    {
        if (!in_array(
            countryCode,
            CountryCode.knownCountryCodes)
        ) {
            throw new InvalidArgumentException(
                'Unknown country code: ' . countryCode
            );
        }
        this.countryCode = countryCode;
    }
}
```

4 Вариант ответа:

```
final class Distance
{
    private int distance;
    private string unit;

    public function __construct(int distance, string unit)
    {
        if (distance <= 0) {
            throw new InvalidArgumentException(
                'distance should be greater than 0'
            );
        }
        this.distance = distance;

        if (!in_array(unit, ['meters', 'feet'])) {
            throw new InvalidArgumentException(
                'Unknown unit: ' unit
            );
        }
        this.unit = unit;
    }
}

final class Run
{
    public function __construct(Distance distance)
    {
        // ...
    }
}
```

5 Вариант ответа:

```
final class PriceRange
{
```

```
    public function __construct(int minimumPrice, int maximumPrice)
    {
        Assertion.greaterThanOrEqual(minimumPrice, 0);
        Assertion.greaterThanOrEqual(maximumPrice, 0);
        Assertion.greaterThan(maximumPrice, minimumPrice);

        this.minimumPrice = miminumPrice;
        this.maximumPrice = maximumPrice;
    }
}
```

6 Правильный ответ: **в**. `User` — это не сервис, а сущность, поэтому ей не должны передаваться зависимости в виде аргументов конструктора или методов-сеттеров. Также она не должна обращаться к зависимости. Вместо этого любая зависимость, необходимая для выполнения задачи, должна передаваться в качестве аргументов метода.

7 Правильный ответ: **б**. Другие варианты — примеры плохой модели проектирования: удаление типов, добавление множества аргументов, из которых использоваться будет лишь один, а также расширение класса, которым вы не управляете и не можете добавить в него необходимое поведение.

8 Правильный ответ: **б**. Геттеры не запрещены, и свойство может иметь значение `null`. Единственное правило — не добавлять геттеры только в целях тестирования.

9 Правильный ответ: **б**. Как только сущность распознает недопустимость передаваемых клиентом данных, она выдаст исключение. Это, в свою очередь, лишит возможности анализировать доступные данные и генерировать списки ошибок.

10 Правильный ответ: **а**. DTO будет принимать предоставляемые данные, если их тип соответствует ожидаемому. Сущность выдаст исключение, как только встретит недопустимые данные.

11 Правильный ответ: **а**. DTO по умолчанию раскрывает все свои данные. Сущности же, наоборот, обычно защищают большинство своих внутренних данных.

4 Изменение объектов

В этой главе:

- ✓ Разница между изменяемыми и неизменяемыми объектами
- ✓ Использование модификаторов методов для изменения состояния объектов или создания измененных копий
- ✓ Сравнение объектов
- ✓ Защита от недопустимых изменений состояния
- ✓ Использование событий для отслеживания изменений в объектах

Как вы уже знаете из предыдущих глав, сервисы необходимо проектировать неизменяемыми. Это значит, что после создания объекта сервиса его должно быть невозможно изменить. Главное преимущество такого подхода в том, что поведение сервиса станет предсказуемым, а сам сервис можно использовать для выполнения задач с различными входными данными.

Итак, сервисы должны быть неизменяемыми. А как насчет других типов объектов: сущностей, значений-объектов или DTO?

4.1. СУЩНОСТИ: ИДЕНТИФИЦИРУЕМЫЕ ОБЪЕКТЫ, КОТОРЫЕ ОТСЛЕЖИВАЮТ ИЗМЕНЕНИЯ И ФИКСИРУЮТ СОБЫТИЯ

Сущности — основные объекты приложений. Они представляют важные концепции предметной области, например резервирование, заказ, счет, продукт, потребитель и т. п. Они моделируют знания, полученные разработчиками об этой области бизнеса. В сущности хранится соответствующая информация и могут быть предложены способы управления этой информацией; сущности могут раскрывать некоторые сведения на основе этой информации. Примером такой сущности может выступать класс `SalesInvoice`, представленный ниже.

Листинг 4.1. Сущность SalesInvoice

```
final class SalesInvoice
{
    /**
     * @var Line[]
     */
    private array lines = [];

    private bool finalized = false;

    public static function create(/* ... */): SalesInvoice ◄── Вы можете создать счет
    {
        // ...
    }

    public function addLine(/* ... */): void ◄──┐
    {                                           │ Вы можете управлять его состоянием,
        if (this.finalized) {                   │ например добавлять в него строки
            throw new RuntimeException(/* ... */);
        }

        this.lines[] = Line.create(/* ... */);
    }

    public function finalize(): void ◄── Вы можете завершить его создание
    {
        this.finalized = true;
        // ...
    }                                           ┐
                                                │ Объект раскрывает некоторую
    public function totalNetAmount(): Money ◄──┘ информацию о себе
    {
        // ...
    }

    public function totalAmountIncludingTaxes(): Money
    {
        // ...
    }
}
```

Сущность может меняться со временем, но она должна оставаться одним и тем же изменяемым объектом. Именно поэтому сущность должна быть идентифицируемой: при ее создании мы присваиваем ей идентификатор.

Листинг 4.2. SalesInvoice получает идентификатор во время вызова конструктора

```
final class SalesInvoice
{
    private SalesInvoiceId salesInvoiceId;

    public static function create(
        SalesInvoiceId salesInvoiceId
    ): SalesInvoice {
        object = new SalesInvoice();

        object.salesInvoiceId = salesInvoiceId;

        return object;
    }
}
```

Идентификатор может использоваться репозиторием сущности для сохранения объекта. Позже мы применим идентификатор для получения объекта из репозитория, после чего объект снова можно будет изменять.

Листинг 4.3. Использование идентификатора для изменения созданной ранее сущности

```
salesInvoiceId = this.salesInvoiceRepository.nextIdentity();
salesInvoice = SalesInvoice.create(salesInvoiceId);
this.salesInvoiceRepository.save(salesInvoice);

salesInvoice = this.salesInvoiceRepository.getBy(salesInvoiceId);
salesInvoice.addLine(/* ... */);
this.salesInvoiceRepository.save(salesInvoice);
```

Для начала создайте SalesInvoice и сохраните ее

Потом снова извлеките ее и внесите в нее изменения

Так как состояние сущности меняется со временем, она является изменяемым объектом. Реализация сущности подчиняется определенным правилам:

- Методы, которые изменяют состояние сущности, должны иметь возвращаемый тип `void`, а их имена должны быть в форме императива (например, `addLine()`, `finalize()`).
- Эти методы должны защищать сущность от недопустимого состояния (например, в `addLine()` должно проверяться, что создание объекта счета еще не завершено).
- Сущность не должна раскрывать все внутренние детали для тестирования того, что происходит внутри объекта. Вместо этого сущность должна вести

журнал изменений и раскрывать внешним клиентам именно его, чтобы другие объекты узнавали, что в объекте изменилось и почему.

В следующем листинге показано, как в `SalesInvoice` ведется журнал изменений, где фиксируются внутренние события предметной области, которые можно получить при вызове метода `recordedEvents()`.

Листинг 4.4. В сущности SalesInvoice ведется журнал внутренних изменений

```
final class SalesInvoice
{
    /**
     * @var object[]
     */
    private array events = [];
    private bool finalized = false;
    public function finalize(): void
    {
        this.finalized = true;

        this.events[] = new SalesInvoiceFinalized(/* ... */);
    }

    /**
     * @return object[]
     */
    public function recordedEvents(): array
    {
        return this.events;
    }
}

salesInvoice = SalesInvoice.create(/* ... */);  ← В тестовом сценарии ...
salesInvoice.finalize();

assertEquals(
     [
        new SalesInvoiceFinalized(/* ... */)
    ],
    salesInvoice.recordedEvents()
);

salesInvoice = this.salesInvoiceRepository.getBy(salesInvoiceId);
salesInvoice.finalize(/* ... */);
this.salesInvoiceRepository.save(salesInvoice);

this.eventDispatcher.dispatchAll(
    salesInvoice.recordedEvents()
);
```

В сервисе можно разрешить прослушивателям событий отвечать на внутренние сохраняемые события

4.2. ОБЪЕКТЫ-ЗНАЧЕНИЯ: ЗАМЕНЯЕМЫЕ, АНОНИМНЫЕ И НЕИЗМЕНЯЕМЫЕ ЗНАЧЕНИЯ

Объекты-значения — это совершенно другая история. Они зачастую гораздо меньше и обладают всего одним или двумя свойствами. В них также могут быть представлены понятия предметной области, и в таком случае они являются частью сущности или находятся в области ее действия. Например, в сущности `SalesInvoice` нам нужны объекты-значения для идентификатора счета, даты создания счета, а также идентификаторов и количества каждого из продуктов. В листинге ниже показана примерная схема использования классов объектов-значений.

Листинг 4.5. Объекты-значения используются сущностью SalesInvoice

```
final class SalesInvoiceId
{
    // ...
}

final class Date
{
    // ...
}

final class Quantity
{
    // ...
}

final class ProductId
{
    // ...
}

final class SalesInvoice
{
    public static function create(
        SalesInvoiceId salesInvoiceId,
        Date invoiceDate
    ): SalesInvoice {
        // ...
    }

    public function addLine(
        ProductId productId,
        Quantity quantity
    ): void {
        this.lines[] = Line.create(
            productId,
            quantity
        );
    }
}
```

Как было показано в предыдущей главе, объекты-значения оборачивают одно или более примитивных значений и их можно создавать, передавая эти значения в конструктор:

```
final class Quantity
{
    public static function fromInt(
        int quantity,
        int precision
    ): Quantity {
        // ...
    }
}

final class ProductId
{
    public static function fromInt(int productId): ProductId
    {
        // ...
    }
}
```

Нет необходимости идентифицировать объекты-значения. Нам не нужны точные подробности об экземпляре, с которым мы работаем, так как не нужно отслеживать изменения, которые вносятся в объект-значение. На самом деле мы вообще не должны изменять объект-значение. Если его требуется преобразовать в другое значение, надо инстанцировать новый объект, содержащий изменение. Например, при сложении двух значений вместо того, чтобы изменять внутреннее значение экземпляра `Quantity`, мы возвращаем новый объект `Quantity`, содержащий значение суммы.

Листинг 4.6. Метод add() возвращает новый экземпляр класса Quantity

```
final class Quantity
{
    private int quantity;
    private int precision;

    private function __construct(
        int quantity,
        int precision
    ) {
        this.quantity = quantity;
        this.precision = precision;
    }

    public static function fromInt(
        int quantity,
        int precision
    ): Quantity {
        return new Quantity(quantity, precision);
```

```
    }

    public function add(Quantity other): Quantity
    {
        Assertion.same(this.precision, other.precision);

        return new Quantity(
            this.quantity + other.quantity,
            this.precision
        );
    }
}

originalQuantity = Quantity.fromInt(1500, 2);

newQuantity = originalQuantity.add(Quantity.fromInt(500, 2));
```

Значение 1500 с двумя знаками после точки представляет собой значение 15.00

Измененное значение представляет собой 15.00 + 5.00 = 20.00

Возвращая новый экземпляр вместо изменения состояния существующего объекта, мы делаем объект-значение `Quantity` неизменяемым.

Объекты-значения не только представляют понятия предметной области. Они могут использоваться в любом месте приложения. Объект-значение — это любой неизменяемый объект, оборачивающий значения примитивного типа.

4.3. ОБЪЕКТЫ ДЛЯ ПЕРЕДАЧИ ДАННЫХ: ПРОСТЫЕ ОБЪЕКТЫ С МИНИМАЛЬНЫМ НАБОРОМ ПРАВИЛ ПРОЕКТИРОВАНИЯ

Еще одним типом объектов, который оборачивает значения примитивного типа, являются *объекты для передачи данных (DTO)*; мы обсуждали их в предыдущей главе. И хотя некоторые предпочитают реализовывать такие объекты как неизменяемые, это может помешать использовать другие свойства DTO. Например, вам может понадобиться заполнять свойства одно за другим на основе данных, предоставляемых пользователем. Вам также не требуется обслуживать такой объект или писать для него модульные тесты, так как он не содержит выраженного поведения (а только данные), поэтому не стоит наделять его большим количеством методов (например, геттерами и сеттерами). В конце концов, вы используете модификатор доступа `public` для свойств. Если в языке есть возможность помечать значения как доступные только для чтения или одноразовой записи (например, ключевое слово `final` в Java), в такой ситуации это будет вполне уместно.

В следующем листинге показан пример DTO класса `CreateSalesInvoice`, в котором также содержатся экземпляры DTO класса `Line`.

Листинг 4.7. DTO-классы с публичными полями

```
final class CreateSalesInvoice
{
    /**
     * @final
     */
    public string date;

    /**
     * @var Line[]
     * @final
     */
    public array lines = [];
}

final class Line
{
    /**
     * @final
     */
    public int productId;

    /**
     * @final
     */
    public int quantity;
}
```

Для DTO не существует столь же универсальных правил проектирования, как для сущностей или объектов-значений. Для последних качество проектирования и целостность данных более важны, чем для DTO. Поэтому правила проектирования в этой главе касаются только сущностей и объектов-значений.

УПРАЖНЕНИЯ

1 Какой тип объекта представлен в классе ниже?

```
final class UserId
{
    private int userId;

    private function __construct(int userId)
    {
        this.userId = userId;
    }

    public static function fromInt(int userId): UserId
    {
        return new UserId(userId);
    }
}
```

а Сущность.

б Объект-значение.

в Объект для передачи данных.

2 Какой тип объекта представлен в классе ниже?

```
final class User
{
    private UserId userId;
    private Username username;
    private bool isActive;

    private function __construct()
    {
    }

    public static function create(
        UserId userId,
        Username username
    ): User {
        user = new User();

        user.userId = userId;
        user.username = username;

        return user;
    }

    public function deactivate(): void
    {
        this.active = false;
    }
}
```

а Сущность.

б Объект-значение.

в Объект для передачи данных.

3 Какой тип объекта представлен в классе ниже?

```
final class CreateUser
{
    public string username;
    public string password;
}
```

а Сущность.

б Объект-значение.

в Объект для передачи данных.

4.4. ОТДАВАЙТЕ ПРЕДПОЧТЕНИЕ НЕИЗМЕНЯЕМЫМ ОБЪЕКТАМ

Так как в число задач сущности входит отслеживание изменений, ей не помешает возможность менять внутренние значения уже после вызова конструктора. На самом деле большинство объектов, которые не являются сущностями, стоит реализовывать в виде неизменяемых объектов-значений. Рассмотрим подробнее, почему лучше проектировать объекты неизменяемыми.

По определению, объект создается, а затем многократно используется в различных ситуациях. Можно передавать объект в качестве аргумента метода или конструктора или присвоить его значению свойства:

```
object = new Foo();

this.someMethod(object);   <-- Передача объекта

this.someProperty = object;   <-- Присвоение значения объекта свойству

return object;   <-- Возможное возвращение объекта
```

Если у одной точки вызова будет ссылка на объект, а другая внесет изменения в некоторых областях объекта, то для первой точки это окажется сюрпризом. Как в таком случае понять, что объект еще можно использовать? Первая сторона может не знать, как обращаться с новым состоянием объекта.

Тем не менее даже в пределах одной точки могут возникать проблемы, связанные с изменяемостью объекта. Взгляните на следующий листинг.

Листинг 4.8. В классе Appointment есть проблемы с изменяемостью

```
final class Appointment
{
    private DateTime time;

    public function __construct(DateTime time)
    {
        this.time = time;
    }

    public function time(): string
    {
        return this.time.format('h:s');
    }

    public function reminderTime(): string
    {
        oneHourBefore = '-1 hour';

        reminderTime = this.time.modify(oneHourBefore);   <-- Здесь изменяется объект, который содержится в свойстве time
```

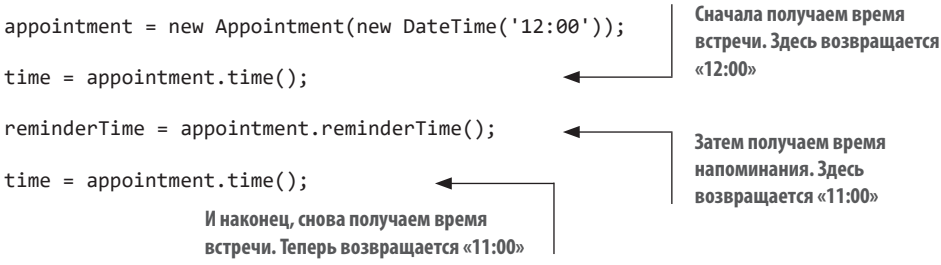
```
        return reminderTime.format('h:s');
    }
}

appointment = new Appointment(new DateTime('12:00'));

time = appointment.time();

reminderTime = appointment.reminderTime();

time = appointment.time();
```



Глядя на этот код, сложно догадаться, почему после запроса времени для напоминания время самой встречи изменилось. Чтобы предотвратить подобные ситуации, объекты, которые не являются сущностями, нужно проектировать неизменяемыми.

4.4.1. Заменяйте значения новыми, а не изменяйте их

Если вы проектируете объекты неизменяемыми, то они становятся похожи на примитивы. Рассмотрим следующий пример:

```
i = 1;
i++;
```

Можно ли сказать, что число 1 изменилось на 2? Нет, здесь можно сказать лишь то, что переменная `i` ранее содержала значение 1, а теперь содержит значение 2. Значения целочисленных переменных сами по себе неизменяемы. Мы их используем, а затем избавляемся от них, но мы можем использовать их снова. Кроме того, передача их в качестве аргументов метода или копирование их в свойства объекта ничем не грозит. Каждый раз, когда нам нужно целое число, мы создаем его из бесконечного набора целых чисел. В памяти компьютера нет общедоступного места, где хранится по одному экземпляру каждого целого числа.

То же касается и объектов, которые реализованы как неизменяемые значения. Они больше не общедоступны. И если нам нужно другое состояние объекта, то мы создаем новый. Это значит, что если неизменяемый объект является значением переменной или свойства, то при желании что-то изменить придется создать новый объект и хранить в нем новое значение переменной или свойства.

Для наглядности представим, что мы реализовали `Year` в виде неизменяемого объекта, обернув целочисленное значение и предоставив удобный метод для получения нового экземпляра `Year`, который будет содержать численное значение следующего года.

Листинг 4.9. Класс Year

```
final class Year
{
    private int year;

    public function __construct(int year)
    {
        this.year = year;
    }

    public function next(): Year
    {
        return new Year(this.year + 1);
    }
}
```

```
year = new Year(2019);

year.next();
assertEquals(new Year(2019), year);

year = year.next();
assertEquals(new Year(2020), year);
```

Здесь ничего не происходит, так как next() не изменяет само значение года

Вместо этого мы должны сохранить значение, возвращаемое методом next()

Если мы будем содержать экземпляр `Year` в качестве значения свойства изменяемого объекта и захотим перейти к следующему году, следует не только вызвать метод `next()`, но и сохранить возвращаемое значение в свойстве, которое содержит значение текущего года, как показано ниже.

Листинг 4.10. Заменяйте значения вместо того, чтобы изменять их

```
final class Journal
{
    private Year currentYear;

    public function closeTheFinancialYear(): void
    {
        // ...

        this.currentYear = this.currentYear.next();
    }
}
```

КАК ОПРЕДЕЛИТЬ, ДОЛЖЕН ЛИ ОБЪЕКТ БЫТЬ НЕИЗМЕНЯЕМЫМ

Если объект является сервисом, тут все понятно: он должен быть неизменяемым. Если это сущность и она должна меняться, то объект должен быть изменяемым. Все остальные объекты должны быть неизменяемыми по причинам, рассмотренным в предыдущем разделе.

На практике в зависимости от типа приложения, над которым вы работаете, все же придется реализовывать некоторые объекты изменяемыми — например, если у приложения интерактивный GUI или вы разработчик игр. Если фреймворк вынуждает вас отойти от правил, иногда придется принять его условия (а иногда стоит отказаться от фреймворка). Просто возьмите за правило создавать объекты неизменяемыми.

УПРАЖНЕНИЕ

4 Следующий класс `ColorPalette` представляет неизменяемый объект, который нужно создать единожды и никогда больше не изменять. К сожалению, текущая реализация не является неизменяемой. Что здесь не так?

```
final class ColorPalette
{
    private Collection colors;

    private function __construct()
    {
        this.colors = new Collection();
    }

    public static function startWith(sRGB color): ColorPalette
    {
        palette = new ColorPalette();

        palette.colors.add(color);

        return palette;
    }

    public function withColorAdded(sRGB color): ColorPalette
    {
        copy = clone this;
        copy.colors = clone this.colors;

        copy.colors.add(color);

        return copy;
    }

    public function colors(): Collection
    {
        return this.colors;
    }
}
```

а Метод `startsWith()` изменяет свойства экземпляра `ColorPalette`, что делает объект изменяемым.

б Метод `colors()` возвращает изменяемую коллекцию, что косвенно делает экземпляр `ColorPalette` изменяемым.

в Метод `withColorAdded()` изменяет исходный экземпляр `ColorPalette`.

4.5. МОДИФИКАТОР НЕИЗМЕНЯЕМОГО ОБЪЕКТА ДОЛЖЕН ВОЗВРАЩАТЬ МОДИФИЦИРОВАННУЮ КОПИЮ

Исходя из наблюдений, неизменяемые объекты могут иметь методы, которые считаются модификаторами, но они не изменяют состояния объекта, на котором вызван метод. Вместо этого метод возвращает копию объекта, но с данными, которые соответствуют целям использования метода. Возвращаемый тип метода должен быть того же класса, что и сам объект, как в методе `next()` из предыдущего примера с классом `Year`.

Существуют два основных способа реализации таких методов. Первый — использование (возможно, приватного) конструктора объекта для создания желаемой копии, как в методе `plus()` в следующем листинге.

Листинг 4.11. Метод plus() возвращает новую копию объекта при помощи существующего конструктора

```
final class Integer
{
    private int integer;

    public function __construct(int integer)
    {
        this.integer = integer;
    }

    public function plus(Integer other): Integer
    {
        return new Integer(this.integer + other.integer);
    }
}
```

Так как в классе `Integer` уже есть конструктор, который принимает `int`-значение, можно добавлять существующие целочисленные значения и передавать получаемое `int` в конструктор класса `Integer`.

Еще один способ, который иногда полезно использовать с неизменяемыми объектами с множеством свойств, — создание копии объекта при помощи оператора `clone` с последующим внесением изменений. См. листинг ниже для метода `withX()`.

Листинг 4.12. Метод withX() использует оператор clone для создания копии

```
final class Position
{
    private int x;
    private int y;

    public function __construct(int x, int y)
    {
        this.x = x;
        this.y = y;
    }

    public function withX(int x): Position
    {
        copy = clone this;

        copy.x = x;

        return copy;
    }
}

position = new Position(10, 20);

nextPosition = position.withX(6);
assertEquals(new Position(6, 20), nextPosition);
```

Следующей позицией будет 4 шага влево (6, 20)

В предыдущем примере метод `withX()` похож на традиционный метод-сеттер, который просто позволяет клиенту заменить значение одного свойства. Это заставляет клиента заранее вычислять, каким должно быть новое значение. Но, как правило, существуют варианты и получше. Старайтесь делать методы-модификаторы более интеллектуальными или хотя бы присваивайте им имена, связанные с предметной областью, а не просто технические. Для этого оцените, как клиенты используют эти методы.

Например, вот клиент метода `withX()`:

```
nextPosition = position.withX(position.x() - 4);
```

Переместиться на 4 шага влево

Так как у `Position` есть метод-модификатор для присвоения нового значения `x`, то этот клиент должен сначала вычислить, какое значение нужно предоставить. Но клиенту нужно не просто изменить значение `x`, ему нужно узнать, какой будет следующая позиция, если сделать четыре шага влево.

Вместо того чтобы перекладывать предварительные вычисления на клиента, можно реализовать эту задачу в самом объекте `Position`. Нужно только предложить более удобный метод-модификатор, например метод `toTheLeft()` в следующем листинге.

Листинг 4.13. Метод toTheLeft() удобнее, чем withX()

```
final class Position
{
    // ...

    public function toTheLeft(int steps): Position
    {
        copy = clone this;

        copy.x = copy.x - steps;

        return copy;
    }
}

position = new Position(10, 20);

nextPosition = position.toTheLeft(4);
assertEquals(new Position(6, 20), nextPosition);

assertEquals(new Position(10, 20), position);
```

Следующая позиция будет (6, 20)

Исходный объект не должен быть изменен

УПРАЖНЕНИЕ

5 Взгляните на следующие классы объектов-значений `DiscountPercentage` и `Money`.

```
final class DiscountPercentage
{
    private int percentage;

    public static function fromInt(int percentage)
    {
        discount = new DiscountPercentage();

        discount.percentage = percentage;

        return discount;
    }

    public function percentage(): int
    {
        return this.percentage;
```

```
        }
    }

    final class Money
    {
        private int amountInCents;

        public static function fromInt(int amountInCents)
        {
            money = new Money();

            money.amountInCents = amountInCents;

            return money;
        }

        public function amountInCents(): int
        {
            return this.amountInCents;
        }
    }
```

Вот как можно использовать `Money` и `DiscountPercentage` для вычисления цены со скидкой:

```
originalPrice = Money.fromInt(2000);  <-- 20.00 евро

discountPercentage = DiscountPercentage.fromInt(10);  <-- Скидка 10 %

discount = (int)round(
    discountPercentage.percentage() / 100)  <-- Вычисление скидки
        * originalPrice.amountInCents()         и ее вычитание
);                                              из исходной цены
discountedPrice = Money.fromInt(
    originalPrice.amountInCents() - discount
);
```

Вместо того чтобы делать вычисления вне объекта `Money`, напишите метод-модификатор под названием `withDiscountApplied()` для класса `Money`, который самостоятельно производит вычисления.

4.6. В ИЗМЕНЯЕМЫХ ОБЪЕКТАХ МЕТОДЫ-МОДИФИКАТОРЫ ДОЛЖНЫ БЫТЬ КОМАНДНЫМИ

Хотя почти все объекты должны быть неизменяемыми, есть и такие, которые должны меняться, а именно сущности. Как мы видели в начале главы, у сущности есть методы, которые позволяют управлять ею.

Рассмотрим другой пример — класс `Player`, у которого есть текущее положение, представленное в значениях `X` и `Y`. Это изменяемый объект: у него есть метод `moveLeft()`, который меняет (то есть заменяет новым) положение игрока. Объект `Position` неизменяемый, а объект `Player` изменяемый.

Листинг 4.14. Player — изменяемый объект, а Position — нет

```
final class Player
{
    private Position position;

    public function __construct(Position initialPosition)
    {
        this.position = initialPosition;
    }

    public function moveLeft(int steps): void
    {
        this.position = this.position.toTheLeft(steps);
    }

    public function currentPosition(): Position
    {
        return this.position;
    }
}
```

На изменяемость объекта указывает оператор присвоения в методе `moveLeft()`: свойство `position` получает новое значение при вызове этого метода. Другой признак — возвращаемый тип `void`. Эти два свойства служат характерными признаками так называемых *командных методов*.

Методы, которые изменяют состояние объекта, всегда должны быть командными. Их имя содержит императив, им разрешается вносить изменения во внутренние структуры данных объекта, и они ничего не возвращают.

4.7. В НЕИЗМЕНЯЕМЫХ ОБЪЕКТАХ МЕТОДЫ-МОДИФИКАТОРЫ ДОЛЖНЫ ИМЕТЬ ДЕКЛАРАТИВНЫЕ ИМЕНА

Методы-модификаторы у изменяемых объектов создаются для того, чтобы изменять состояние объекта, что прекрасно соотносится с привычными характеристиками командных методов. Но для методов неизменяемых объектов применяется другое соглашение.

Представьте ту же реализацию `Position`, что мы видели выше, с тем лишь отличием, что метод `toTheLeft()` носит название `moveLeft()`.

Листинг 4.15. moveLeft() вместо toTheLeft()

```
final class Position
{
    // ...

    public function moveLeft(int steps): Position
    {
        // ...
    }
}
```

Исходя из того что методы-модификаторы в изменяемых объектах — это командные методы, `moveLeft()` может сбивать с толку: у него императивное имя (`moveLeft()`), но возвращаемый тип не `void`. И чтобы узнать, меняет ли метод состояние объекта, придется разбираться в реализации.

Чтобы создать хорошее название для методов-модификаторов в неизменяемых объектах, следуйте схеме: «Мне нужно это же.., но...». В случае с `Position` этот пример превращается в «Мне нужна эта позиция, но на `n` шагов левее», поэтому название `toTheLeft()` подходит больше.

Листинг 4.16. toTheLeft() — более подходящее имя

```
final class Position
{
    // ...

    public function toTheLeft(int steps): Position
    {
        // ...
    }
}
```

Следуя этому образцу, можно использовать слово with или причастие в прошедшем времени. Например: «Мне нужно это количество, но умноженное (multiplied) на `n`». Или: «Мне нужен этот ответ, но с заголовком `Content-Type: text/html`». Это декларативные имена: они не говорят, что делать, но «объявляют» (declare) результат действия.

При поиске подходящих названий также стремитесь соответствовать предметной области, используйте высокоуровневые имена вместо обобщенных технических имен нижнего уровня. Например, мы выбрали имя `toTheLeft()` вместо `withXDecreasedBy()`, что демонстрирует различие в уровне абстракции.

УПРАЖНЕНИЯ

6 У объекта есть следующий метод:

```
public setPassword(string plainTextPassword): void
```

Какой ожидается объект: изменяемый или неизменяемый?

а Изменяемый.

б Неизменяемый.

7 У объекта есть следующий метод:

```
public withPassword(string plainTextPassword): User
```

Какой ожидается объект: изменяемый или неизменяемый?

а Изменяемый.

б Неизменяемый.

8 У объекта есть следующий метод:

```
withPassword(string plainTextPassword): void
```

Какой ожидается объект: изменяемый или неизменяемый?

а Изменяемый.

б Неизменяемый.

4.8. СРАВНИВАЙТЕ ОБЪЕКТЫ ЦЕЛИКОМ

Для изменяемых объектов можно писать, например, такие тесты.

Листинг 4.17. Модульный тест для moveLeft()

```
public function it_can_move_to_the_left(): void
{
    position = new Position(10, 20);
    position.moveLeft(4);
    assertSame(6, position.x());
}
```

Как мы уже говорили, такой тест обычно вынуждает добавлять в класс геттеры. Они будут нужны только для тестов; других клиентов, заинтересованных в геттерах, может не быть.

В случае с неизменяемыми объектами можно прибегнуть к проверке, которая позволяет объекту не раскрывать детали реализации.

Листинг 4.18. Модульный тест для метода toTheLeft()

```
public function it_can_move_to_the_left(): void
{
    position = new Position(10, 20);
    nextPosition = position.toTheLeft(4);
    assertEquals(new Position(6, 20), nextPosition);
}
```

`assertEquals()` будет использовать рекурсивный метод, который тестирует равность свойств объектов, а также объектов в этих свойствах и т. п. `assertEquals()`, таким образом, предотвращает ситуацию, когда некоторые неявные свойства объектов-значений могут стать препятствием для сравнения двух объектов.

4.9. ПРИ СРАВНЕНИИ НЕИЗМЕНЯЕМЫХ ОБЪЕКТОВ ПРОВЕРЯЙТЕ, ЧТО ОБЪЕКТЫ РАВНЫ, А НЕ ОДИНАКОВЫ

В примере ниже показано, как класс `Position` из предыдущего примера можно использовать в (изменяемом) классе `Player`.

Листинг 4.19. Класс Player

```
final class Player
{
    private Position position;

    public function __construct(Position initialPosition)
    {
        this.position = initialPosition;
    }

    public function moveLeft(int steps): void
    {
        this.position = this.position.toTheLeft(steps);
    }

    public function currentPosition(): Position
    {
        return this.position;
    }
}
```

Тест для метода `moveLeft()` может быть таким:

Листинг 4.20. Модульный тест для moveLeft()

```
function the_player_starts_at_a_position_and_can_move_left(): void
{
    initialPosition = new Position(10, 20);
    player = new Player(initialPosition);

    assertSame(initialPosition, player.currentPosition());

    player.moveLeft(4);

    assertEquals(new Position(6, 20), player.currentPosition());
}
```

Достаточно использовать assertSame() — объект Position здесь тот же, что был изначально передан

Здесь нужно использовать assertEquals()

При сравнении неизменяемых объектов тесты не должны строиться на том, что объекты ссылаются на одну и ту же область памяти. На самом деле имеет значение только то, что они сами собой представляют. При сравнении целых чисел мы не сравниваем их ссылки на объекты. Мы просто задаем вопрос: равны ли их значения? Поэтому для сравнения объектов всегда необходимо использовать `assertEquals()`.

Бывает, что нужно сравнить два объекта в коде готового продукта, а не в тесте. В этом случае не получится использовать `assertEquals()`. Действия будут зависеть от используемого языка. В некоторых языках, например Java и C#, есть встроенные механизмы для сравнения объектов. Объекты в этих языках наследуют метод `equals()` от общего класса `Object`, и можно переопределить этот метод собственной логикой сравнения. Если вы используете PHP, имитируйте этот подход. Добавьте метод `equals()` в объект, который сравнивает данные, содержащиеся в обоих объектах, как показано в листинге ниже.

Листинг 4.21. Метод equals() помогает сравнить два объекта Position

```
final class Position
{
    // ...

    public function equals(Position other): bool
    {
        return this.x == other.x && this.y == other.y;
    }
}
```

Тем не менее большинству объектов-значений не нужен специальный метод `equals()`, и определенно не стоит бездумно реализовывать его для каждого неизменяемого объекта. Правило геттеров также применимо к методу `equals()`: добавляйте этот метод, только если его будет использовать другой клиент, кроме теста. Кроме того, по возможности не называйте объект `other`. И в целом клиенты не должны пытаться сравнивать объект `Position` с чем-то, что не является объектом `Position`.

УПРАЖНЕНИЯ

9 Как сравнить два объекта-значения в модульном тесте?

- **а** Сравнить возвращаемые значения их геттеров.
- **б** Воспользоваться специальной функцией сравнения объектов, например `assertEquals()`.
- **в** Сравнить ссылки на объекты (при помощи ==).
- **г** Вызвать метод объекта `equals()`.

10 Как сравнить два объекта-значения в коде готового продукта?

- **а** Сравнить возвращаемые значения их геттеров.
- **б** Воспользоваться специальной функцией сравнения объектов, например `assertEquals()`.
- **в** Сравнить ссылки на объекты (при помощи ==).
- **г** Вызвать метод объекта `equals()`.

4.10. ВЫЗОВ МЕТОДА-МОДИФИКАТОРА ДОЛЖЕН ВСЕГДА ОСТАВЛЯТЬ ДЕЙСТВИТЕЛЬНЫЙ ОБЪЕКТ

Когда мы говорили о создании объектов, мы обсуждали такие понятия, как «значимые данные» и «инварианты предметной области». Эти же понятия применимы и к методам-модификаторам, и не только для неизменяемых объектов, но и для изменяемых.

Метод-модификатор должен проверять, что клиент предоставляет данные, имеющие смысл, а также учитывать инварианты предметной области. Это делается так же, как и в конструкторе: при помощи проверок предоставляемых значений аргументов. Таким образом можно защитить объект от недопустимого состояния. Например, рассмотрим метод `add()`.

Листинг 4.22. TotalDistanceTraveled не принимает отрицательные значения расстояния

```
final class TotalDistanceTraveled
{
    private int totalDistance = 0;

    public function add(int distance): TotalDistanceTraveled
    {
```

```
        Assertion.greaterOrEqualThan(
            distance,
            0,
            'You cannot add a negative distance'
        );

        copy = clone this;
        copy.totalDistance += distance;

        return copy;
    }
}
totalDistanceTravelled = new TotalDistanceTraveled();
expectException(
    InvalidArgumentException.className,
    'distance',
    function () use (totalDistanceTravelled) {
        totalDistanceTravelled.add(-10);
    }
);
```

Если метод-модификатор не клонирует, а снова использует конструктор класса, можно использовать уже имеющуюся в нем логику проверок. На самом деле это хорошая причина не использовать `clone`, а всегда идти через конструктор.

Например, рассмотрим класс `Fraction`, который представляет доли целого числа (например, `1/3`, `2/5`). Структура дроби такая: `[numerator]/[denominator]`. Оба эти значения могут быть любым целым числом, только знаменатель не может быть равен `0`. Конструктор уже учитывает это правило, поэтому методу-модификатору `withDenominator()` нужно только перенаправить текущий вызов к методу-конструктору, и правило будет проверяться также для аргументов метода `withDenominator()`.

Листинг 4.23. withDenominator() использует логику проверок из конструктора

```
final class Fraction
{
    private int numerator;
    private int denominator;

    public function __construct(int numerator, int denominator)
    {
        Assertion.notEq(
            denominator,
            0,
            'The denominator of a fraction cannot be 0'
        );

        this.numerator = numerator;
        this.denominator = denominator;
    }
```

```
    public function withDenominator(newDenominator): Fraction
    {
        return new Fraction(this.numerator, newDenominator);
    }
}

fraction = new Fraction(1, 2);

expectException(
    InvalidArgumentException.className,
    'denominator',
    function () use (fraction) {
        fraction.withDenominator(0);
    }
);
```

Перенаправление текущего вызова в конструктор также запустит любые его проверки

УПРАЖНЕНИЕ

11 Что не так со следующей реализацией объекта Range?

```
final class Range
{
    private int minimum;
    private int maximum;

    private function __construct(int minimum, int maximum)
    {
        Assertion.greaterThan(maximum, minimum);

        this.minimum = minimum;
        this.maximum = maximum;
    }

    public static function fromIntegers(
        int minimum,
        int maximum
    ): Range {
        return new Range(minimum, maximum);
    }

    public function withMinimum(int minimum): Range
    {
        copy = clone this;
        copy.minimum = minimum;

        return copy;
    }

    public function withMaximum(int maximum): Range
    {
```

```
            Assertion.greaterThan(maximum, this.minimum);

            copy = clone this;
            copy.maximum = maximum;

            return copy;
        }
    }
```

а `withMinimum()` и `withMaximum()` создают неполные копии объекта `Range`.

б Ни в одном методе-модификаторе не проверяется правило «максимальное значение должно быть больше минимального».

4.11. МЕТОД-МОДИФИКАТОР ДОЛЖЕН ПРОВЕРЯТЬ, ЧТО ЗАПРАШИВАЕМОЕ ИЗМЕНЕНИЕ СОСТОЯНИЯ ДОПУСТИМО

Вызов метода-модификатора для объекта часто предполагает изменение свойств объекта. Для изменяемых объектов, например сущностей, такое изменение в состоянии объекта представляет собой *переход состояния*. Переход состояния может открыть новые возможности или лишить прежних возможностей.

Например, рассмотрим следующий класс, `SalesOrder`. Как только объект класса получит статус «доставлено», его нельзя будет отменить, так как этот переход состояния не будет иметь никакого смысла с точки зрения бизнеса. То же касается и присвоения статуса «доставлено» заказу, который был отменен. После отмены заказ не должен доставляться.

Листинг 4.24. SalesOrder не позволяет производить некоторые переходы

```
final class SalesOrder
{
    // ...

    public function markAsDelivered(Timestamp deliveredAt): void
    {
        /*
         * Не должно быть возможности доставить заказ
         * после его отмены.
         */
    }

    public function cancel(Timestamp cancelledAt): void
    {
```

```
        /*
         * Не должно быть возможности отменить заказ
         * после его доставки.
         */
    }

    // и т. д...
}
```

Убедитесь, что каждый из методов не позволяет осуществлять недопустимые переходы состояний. Проверяйте это при помощи модульных тестов, как в листинге ниже.

Листинг 4.25. Модульный тест для случая отмены доставленного заказа

```
public function a_delivered_sales_order_can_not_be_cancelled(): void
{
    deliveredSalesOrder = /* ... */;
    deliveredSalesOrder.markAsDelivered(/* ... */);

    expectException(
        LogicException.className,
        'delivered',
        function () use (deliveredSalesOrder) {
            deliveredSalesOrder.cancel();
        }
    );
}
```

Подходящим исключением для такого случая будет `LogicException`, но можно создать и свой тип исключения, например `CanNotCancelOrder`.

Листинг 4.26. Если SalesOrder уже отменен, необходимо игнорировать запрос

```
public function cancel()
{
    if (this.status.equals(Status.cancelled())) {
        return;
    }

    // ...
}
```

4.12. ИСПОЛЬЗУЙТЕ ЗАПИСЬ ВНУТРЕННИХ СОБЫТИЙ ДЛЯ ПРОВЕРКИ ИЗМЕНЯЕМЫХ ОБЪЕКТОВ

Мы уже видели, как тестирование конструктора приводит к появлению большего, чем необходимо, количества геттеров в объектах, и все для того, чтобы проверить, что передается в него и что в результате получается. Это не соответствует

концепции объекта, которая гласит, что информация и детали реализации объекта должны быть скрыты. То же самое касается и методов-модификаторов.

Для тестирования метода `moveLeft()` изменяемого объекта `Player`, о котором речь шла выше, есть несколько вариантов. Первый — использовать геттер для проверки, что текущее положение после смещения влево соответствует ожидаемому.

Листинг 4.27. Можно проверить, что текущее положение соответствует ожидаемому

```
public function it_can_move_left(): void
{
    player = new Player(new Position(10, 20));
    player.moveLeft(4);

    assertEquals(new Position(6, 20), player.currentPosition());
}
```

Другой, более прямолинейный вариант — проверить, что состояние всего объекта соответствует ожиданиям.

Листинг 4.28. Можно сравнить весь объект Player с ожидаемым

```
public function it_can_move_left(): void
{
    player = new Player(new Position(10, 20));
    player.moveLeft(4);

    assertEquals(new Player(new Position(6, 20)), player);
}
```

Этот вариант не так уж плох: здесь хотя бы не нужен геттер для получения текущего состояния. Основная проблема в том, что тест затрагивает множество данных и нельзя просто добавлять новое поведение в объект `Player` — вдобавок придется модифицировать и сам тест (в частности, если аргументы конструктора будут добавляться со временем).

Еще один вариант — немного изменить метод `moveLeft()` и сделать так, чтобы он возвращал текущее положение.

Листинг 4.29. moveLeft() возвращает новый объект Position

```
final class Player
{
    public function moveLeft(): Position
    {
        this.position = this.position.toTheLeft(steps);

        return this.position;
    }
}
```

```
player = new Player(new Position(10, 20));
currentPosition = player.moveLeft(4);

assertEquals(new Position(6, 20), currentPosition);
```

Уже лучше, но это нарушает правило, согласно которому метод-модификатор в изменяемом объекте должен быть командным, а значит, иметь возвращаемый тип `void`. К тому же этот тест не доказывает, что `Player` действительно переместился в ожидаемое положение. Рассмотрим, например, следующую реализацию метода `moveLeft()`, в которой тест в листинге 4.29 тоже прошел бы. В нем возвращается правильный объект `Position`, но не изменяется значение свойства объекта `Player`.

Листинг 4.30. Неправильная реализация, при которой тест успешно проходит

```
public function moveLeft(): Position
{
    return this.position.toTheLeft(steps);
}
```

Лучший способ тестирования изменяемых объектов — запись внутренних событий объекта, чтобы их можно было изучить позже. Эти события будут выступать в качестве журнала изменений, которые произошли с объектом. События будут простыми объектами-значениями, и их можно создавать в любом количестве. В листинге ниже класс `Player` переписан так, чтобы производилась запись событий `PlayerEvents`, а раскрывались они в методе `recordedEvents()`.

Листинг 4.31. При изменении состояния Player записывает событие

```
final class Player
{
    private Position position;

    private array events = [];

    public function __construct(Position initialPosition)
    {
        this.position = initialPosition;
    }

    public function moveLeft(int steps): void
    {
        nextPosition = this.position.toTheLeft(steps);

        this.position = nextPosition;

        this.events[] = new PlayerMoved(nextPosition);
    }

    public function recordedEvents(): array
```

После смещения влево мы записываем событие, которое позже можно использовать для выявления того, что произошло в объекте Player

```
    {
        return this.events;
    }
}

player = new Player(new Position(10, 20));

player.moveLeft(4);

assertEquals(
     [
        new PlayerMoved(new Position(6, 20))
    ],
    player.recordedEvents()
);
```

Создание нового Player с заданным исходным положением

Перемещение на 4 шага влево

Проверка того, что положение игрока сместилось, при помощи сравнения записанных и ожидаемых событий

Есть и другие интересные варианты, например запись событий только тогда, когда изменения действительно произошли. Например, игроку разрешается сделать 0 шагов. В этом случае игрок не перемещается, и нет надобности вызывать метод `moveLeft()` для записи события.

Листинг 4.32. Можно записывать события выборочно

```
public function moveLeft(int steps): void
{
    if (steps == 0) {
        return;
    }

    nextPosition = this.position.toTheLeft(steps);

    this.position = nextPosition;

    this.events[] = new PlayerMoved(nextPosition);
}
```

Не выдавайте здесь исключение, но и не записывайте событие

Со временем `assertEquals([/* ... */], player.recordedEvents())` может показаться недостаточно гибким, из-за того что при каждом изменении реализации объекта `Player` существующие тесты будут завершаться ошибкой. Например, посмотрим, что произойдет, если записать еще одно событие: момент, когда игрок занимает исходное положение.

Листинг 4.33. Player также записывает событие PlayerTookInitialPosition

```
final class PlayerTookInitialPosition
{
    // ...
}

final class Player
{
```

```
    private events;

    public function __construct(Position initialPosition)
    {
        this.position = initialPosition;

        this.events[] = new PlayerTookInitialPosition(
            initialPosition
        );
    }
}
```

Это вызовет ошибку теста, который мы составили для перемещения влево.

Листинг 4.34. Существующий тест метода moveLeft(), который теперь будет завершаться ошибкой

```
public function it_can_move_left(): void
{
    player = new Player(new Position(10, 20));
    player.moveLeft(4);

    assertEquals(
        [
            new PlayerMoved(new Position(6, 20))
        ],
        player.recordedEvents()
    );
}
```

Эта проверка будет завершаться ошибкой, так как конструктор теперь записывает событие PlayerTookInitialPosition, которое также будет возвращаться методом recordedEvents()

Проверка того, что список записанных событий *содержит* ожидаемое событие, сделает тест проходимым.

Листинг 4.35. assertContains() может сравнивать записанные события

```
public function it_can_move_left(): void
{
    player = new Player(new Position(10, 20));
    player.moveLeft(4);

    assertContains(
        new PlayerMoved(new Position(6, 20)),
        player.recordedEvents()
    );
}
```

Взгляните на альтернативную реализацию метода `moveLeft()` в следующем листинге, где событие записывается, но положение игрока не обновляется в свойстве `position`. Тест в листинге 4.35 в этом неверно реализованном случае тоже будет пройден успешно.

Листинг 4.36. moveLeft() только записывает событие, но тест все же пройдет

```
final class Player
{
    //...

    public function moveLeft(int steps): void
    {
        this.events[] = new PlayerMoved(nextPosition);
    }
}
```

На самом деле реализацию нельзя считать неверной. Если тест проходит, но готовый код ошибочный, то в поведении объекта, должно быть, есть нечто, не учтенное в тесте. Поэтому в некотором смысле здесь неверен именно тест. Чтобы его исправить, необходимо убедиться, что некоторый другой тест проверяет обновление самого свойства `position` объекта `Player`. Если достаточно веской причины для этого нет, то свойство `position` можно спокойно удалить. Поведение объекта от этого заметно не изменится.

НЕ ЛИШНЕЕ ЛИ ЭТО — СОЗДАВАТЬ СОБЫТИЯ ДЛЯ КАЖДОГО ИЗМЕНЯЕМОГО ОБЪЕКТА?

Как мы уже говорили в начале главы, почти все объекты — неизменяемые. Те немногие объекты, которые изменяются, — это сущности. Это те объекты, для которых есть смысл создавать события (они называются событиями предметной области). Поэтому на практике добавление поддержки записи событий — это не лишнее, а совершенно естественное требование.

Такая возможность в любом случае будет полезной, так как события — это хороший способ отвечать на изменения в объектах предметной области. Другим ответом может быть внесение еще большего количества изменений или использование данных о событиях для обеспечения поиска в поисковых системах, построения моделей чтения или оперативного сбора сведений, полезных бизнесу.

Единственная информация объекта `Player`, которая раскрывается его клиентам, — это список внутренних событий предметной области. Поэтому узнать текущее положение игрока не так легко. На практике же это понадобится только в том случае, если необходимо отобразить текущее положение игрока на экране. Мы вернемся к получению информации из объектов в главе 6.

УПРАЖНЕНИЕ

12 Как лучше узнать в модульном тесте, был ли финализирован объект `SalesInvoice`, инстанцированный из класса ниже?

```
final class SalesInvoice
{
    private string isFinalized = false;

    // ...

    public function finalize()
    {
        this.isFinalized = true;
    }
}
```

а Добавить метод `isFinalized() : bool` в класс `SalesInvoice` и вызвать его до и после вызова `finalize()`, чтобы выяснить, как метод справляется со своей работой.

б Не добавлять геттер, а использовать рефлексию, чтобы извлекать информацию о приватных свойствах.

в Собирать события предметной области внутри сущности и затем анализировать, чтобы выяснить, был ли действительно финализирован `SalesInvoice`.

г Отправлять события предметной области и настроить прослушиватель событий в модульном тесте, отслеживающий, был ли финализирован `SalesInvoice`.

4.13. НЕ РЕАЛИЗУЙТЕ ТЕКУЧИЕ ИНТЕРФЕЙСЫ В ИЗМЕНЯЕМЫХ ОБЪЕКТАХ

Объект реализован с текучим интерфейсом, если его метод-модификатор возвращает `this`. Если у объекта есть текучий интерфейс, можно вызывать на нем метод за методом без необходимости заново использовать переменную объекта.

Листинг 4.37. QueryBuilder позволяет использовать текучий интерфейс

```
queryBuilder = QueryBuilder.create()
    .select(/* ... */)
    .from(/* ... */)
    .where(/* ... */)
    .orderBy(/* ... */);
```

В то же время текучий интерфейс может затруднять понимание, на каком из объектов вызывается метод. Если `QueryBuilder` — неизменяемый объект, то это не имеет значения. Но откуда мы знаем, что он неизменяемый? Из сигнатур методов класса `QueryBuilder` в следующем листинге этого не выяснить.

Листинг 4.38. Сигнатуры методов QueryBuilder

```
final class QueryBuilder
{
    public function select(/* ... */): QueryBuilder
    {
        // ...
    }

    public function from(/* ... */): QueryBuilder
    {
        // ...
    }

    // ...
}
```

Изменяют ли эти методы состояние объекта, на котором они вызываются, или же возвращают модифицированную копию? Или и то и другое?

Исходя из того что эти сигнатуры методов выглядят как модификаторы в неизменяемых объектах, можно предположить, что `QueryBuilder` — неизменяемый и что любую часть объекта `QueryBuilder` просто и безопасно использовать неоднократно, как в листинге ниже.

Листинг 4.39. Многократное использование частей экземпляра QueryBuilder

```
queryBuilder = QueryBuilder.create();

qb1 = queryBuilder
    .select(/* ... */)
    .from(/* ... */)
    .where(/* ... */)
    .orderBy(/* ... */);

qb2 = queryBuilder
    .select(/* ... */)
    .from(/* ... */)
    .where(/* ... */)
    .orderBy(/* ... */);
```

Но затем выясняется, что `QueryBuilder` — изменяемый, это можно заметить по следующей реализации метода `where()`.

Листинг 4.40. Реализация QueryBuilder.where()

```
public function where(string clause, string value): QueryBuilder
{
    this.whereParts[] = clause;
```

```
    this.values[] = value;

    return this;
}
```

Этот метод выглядит как модификатор неизменяемого объекта, но на самом деле это обычный командный метод. И, сбивая с толку, он возвращает текущий экземпляр объекта после его модификации. Чтобы избежать подобных ошибок, не наделяйте изменяемые объекты текучими интерфейсами. `QueryBuilder` все равно лучше делать неизменяемым. Это не позволит передавать клиентам объект в неопределенном состоянии. В листинге ниже показана альтернативная реализация `where()`, в которой `QueryBuilder` делается неизменяемым.

Листинг 4.41. Реализация where(), которая поддерживает неизменяемость объекта

```
public function where(string clause, string value): QueryBuilder
{
    copy = clone this;

    copy.whereParts[] = clause;
    copy.values[] = value;

    return copy;
}
```

Неизменяемые объекты легко поддерживают текучие интерфейсы. По сути, можно смело утверждать, что использование методов-модификаторов так, как описано в этой главе, уже предоставляет доступ к текучему интерфейсу, поскольку каждый такой метод будет возвращать модифицированную копию объекта. Это позволяет выстраивать цепочки вызовов методов в той же манере, как при обычной поддержке текучих интерфейсов (см. следующий листинг).

Листинг 4.42. Метод-модификатор в неизменяемом объекте формирует текучий интерфейс

```
position = Position.startAt(10, 5)
    .toTheLeft(4)
    .toTheRight(2);
```

УПРАЖНЕНИЕ

13 Взгляните на следующий пример сущности класса `Product`. В чем проблема с его методом `setPrice()`?

```
final class Product
{
    // ...
```

```
    public function setPrice(Money price): Product
    {
        // ...
    }
}
```

а Клиент не знает, является ли возвращаемое значение исходным объектом или же его копией.

б Будучи сущностью, `Product` должен быть представлен неизменяемым объектом, но `setPrice()` позволяет модифицировать объект.

в Метод выглядит как метод-модификатор у неизменяемого объекта, но `setPrice()` — это не декларативное, а императивное название.

«В СТОРОННЕЙ БИБЛИОТЕКЕ ОБНАРУЖИЛИСЬ ПРОБЛЕМЫ С ПРОЕКТИРОВАНИЕМ ОБЪЕКТОВ. ЧТО МОЖНО СДЕЛАТЬ?»

Пример с `QueryBuilder` в этом разделе основан на реальном классе `QueryBuilder` из библиотеки Doctrine DBAL (github.com/doctrine/dbal/blob/2.12.x/lib/Doctrine/DBAL/Query/QueryBuilder.php). Это просто один из примеров класса, который не соответствует всем правилам этой книги. Вы наверняка встретитесь и с другими такими классами (в стороннем коде и в коде самого проекта). Но что же с ними делать?

В зависимости от того, как используют классы, можно поискать компромиссы. Например, вы используете плохо спроектированный класс только в рамках своих методов или создаете его экземпляры и передаете их различным методам или даже объектам. В случае с `QueryBuilder` вполне вероятно, что он будет актуален только для методов репозитория. Это значит, что он не может быть применен в других частях приложения, и вы избежите риска встретить его в самом проекте. Поэтому, даже несмотря на проблемы с проектированием класса `QueryBuilder`, переписывать или исправлять его не потребуется.

Бывают и другие случаи с обманчивыми объектами, когда, например, непонятно, изменяемые они или же нет. Примером может послужить класс `DateTime` в PHP или устаревший класс `java.util.Date` в Java. Для них уже существуют неизменяемые альтернативы, но до того, как они появились, было принято копировать эти объекты перед тем, как с ними что-то делать, или создавать собственные неизменяемые объекты-обертки. Это гарантировало, что изменяемые объекты никогда не выйдут за ожидаемые пределы и не будут изменяться другими клиентами, что привело бы к появлению в приложении ошибок, связанных с состоянием.

ЗАКЛЮЧЕНИЕ

- Старайтесь делать выбор в пользу объектов, которые не могут изменяться после создания. Если нужно разрешить внесение изменений, сначала делайте копию, а затем вносите изменения. Присваивайте методам, которые будут это делать, декларативные имена, а также реализуйте действительно полезное поведение вместо простого разрешения изменять значения свойств. Убедитесь, что после вызова метода-модификатора объект остается действительным. Для этого нужно проверять, что через аргументы принимаются только правильные данные и объект не совершает недопустимых переходов состояния.
- У изменяемых объектов — таких как сущности — методы-модификаторы должны иметь возвращаемый тип `void`. Изменения, которые могут возникнуть в таких объектах, можно выявить при помощи анализа внутренних записанных событий. В отличие от неизменяемых объектов, в изменяемых объектах нельзя использовать текучие интерфейсы.

ОТВЕТЫ К УПРАЖНЕНИЯМ

1 Правильный ответ: **б**.

2 Правильный ответ: **а**.

3 Правильный ответ: **в**.

4 Правильный ответ: **б**. `startWith()` — это конструктор, а для конструктора изменять создаваемый экземпляр вполне приемлемо. `withColorAdded()` изменяет не исходный экземпляр `ColorPalette`, а его копию.

5 Вариант ответа:

```
final class Money
{
    // ...

    public function withDiscountApplied(
        DiscountPercentage discountPercentage
    ): Money {
        discount = (int)round(
            (discountPercentage.percentage() / 100)
            * this.amountInCents()
        );

        return Money.fromInt(
            this.amountInCents() - discount
        );
    }
}
```

6 Правильный ответ: **а**. Если бы это был неизменяемый объект, то у него был бы метод-модификатор, возвращающий измененный экземпляр объекта.

7 Правильный ответ: **б**. Если бы это был изменяемый объект, то у него был бы метод-модификатор с возвращаемым типом `void`.

8 Правильный ответ: **а**. Стоит признать, что это неоднозначный метод, так как в нем смешаны декларативный стиль именования с возвращаемым типом командного метода (`void`).

9 Правильный ответ: **б**. Сравнение результатов геттеров сделает тест слишком связанным с объектом-значением. Кроме того, нельзя сравнивать ссылки на значения, так как у объектов они разные. Также нельзя полагаться на стандартный метод `equals()` при сравнении объектов, поскольку нет необходимости вообще сравнивать объекты-значения в рабочем коде — не нужно добавлять этот метод только для целей тестирования.

10 Правильный ответ: **г**. См. ответ к упражнению 9, но в этом случае, по-видимому, есть явная необходимость в сравнении объектов-значений в рабочем коде. Поэтому рекомендуется добавить метод `equals()`.

11 Правильный ответ: **б**. С первого раза неочевидно, но `withMinimum()` и `withMaximum()` создают полноценные копии объекта `Range`. Каждый из методов только перезаписывает значение для одного свойства (`minimum` или `maximum`). Проблема в том, что в `withMinimum()` нет проверки, которая есть в `withMaximum()`, что в дальнейшем может позволить значению `minimum` стать больше значения `maximum`.

12 Правильный ответ: **в**. Нельзя добавлять геттер только ради тестирования; также нельзя раскрывать внутреннее состояние объектов. Вместо этого записывайте события предметной области, чтобы узнавать, что происходит внутри сущности, и анализируйте эту информацию. К тому же немедленно отправлять эти события клиенту не требуется.

13 Правильный ответ: **а** и **в**. Сущность не должна быть неизменяемым объектом.

5 Использование объектов

В этой главе:

- ✓ Применение шаблонов для написания методов
- ✓ Проверка аргументов методов и возвращаемых значений
- ✓ Работа с ошибками внутри методов

После инициализации объекта он готов к использованию. Объекты обладают полезным поведением: они могут предоставлять информацию и совершать работу. В обоих случаях это поведение будет реализовано в виде методов объекта.

Перед тем как обсуждать правила проектирования, специфичные для извлечения информации или для выполнения задач, поговорим про общее для всех методов — шаблон их реализации.

5.1. ШАБЛОН РЕАЛИЗАЦИИ МЕТОДОВ

При проектировании метода всегда используйте следующий шаблон.

Листинг 5.1. Шаблон для создания методов

```
[scope] function methodName(type name, ...): void|[return-type]
{
    [preconditions checks]

    [failure scenarios]

    [happy path]

    [postcondition checks]

    [return void|specific-return-type]
}
```

5.1.1. Проверка предусловий

Первый шаг — проверка, что предоставленные клиентом аргументы корректны и их можно использовать для решения задачи. Выполняйте столько проверок, сколько нужно, и выдавайте исключения, когда что-то идет не так.

Проверка предусловий выглядит так:

```
if (/* предусловие не выполнено */) {
    throw new InvalidArgumentException(/* ... */);
}
```

Как мы уже говорили, для таких проверок можно использовать стандартные функции.

```
Assertion.inArray(value, ['allowed', 'values']);
```

Некоторые из этих проверок нужны тогда, когда в системе типов языка программирования не хватает функций. Например, в PHP есть тип `array`, но нет возможности указать, что массив может содержать только объекты определенного типа. Для этого надо будет добавить утверждение:

```
Assertion.allIsInstanceOf(value, EventListener.className);
```

Другие проверки будут анализировать содержимое аргументов и предупреждать клиента, если, например, предоставленное значение выходит за пределы допустимого диапазона:

```
Assertion.greaterThan(value, 0);
```

Если все проверки проходят успешно, это значит, что мы принимаем аргументы как они есть. Проверки предусловий все же остаются поверхностными, так как они проверяют только наличие очевидных проблем в значениях.

СОЗДАВАЙТЕ НОВЫЕ ТИПЫ, ЧТОБЫ ИЗБАВИТЬСЯ ОТ ПРОВЕРКИ ПРЕДУСЛОВИЙ

Большинство проверок будут выполняться, чтобы проверить аргументы примитивного типа (`int`, `string` и т. д.). Как мы видели в разделе 3.5, зачастую имеет смысл создавать объекты-обертки для таких примитивных значений и перемещать проверки этих значений в конструктор созданных объектов.

```
// Было:

public function sendConfirmationEmail(string emailAddress): void
{
    Assertion.email(emailAddress);
    // ...
}

// Стало:

final class EmailAddress
{
    private string emailAddress;

    public function __construct(string emailAddress)
    {
        Assertion.email(emailAddress);
        this.emailAddress = emailAddress;
    }
}

public function sendConfirmationEmail(
    EmailAddress emailAddress
): void {
    // no need to validate emailAddress anymore
}
```

Такого рода рефакторинг называют заменой примитива объектом[1].

В зависимости от языка программирования реализация метода может допускать передачу аргумента `null` вместо настоящего объекта `EmailAddress`. В таком случае убедитесь, что у вас есть проверка на `null`-аргументы (или на выдачу великого и ужасного `NullPointerException`). Постарайтесь использовать компилятор. В Java, например, для этого существует фреймворк (httcheckerframework.org).

5.1.2. Сценарии появления ошибок

Даже если значения выглядят правильными и проходят проверки предусловий, это не значит, что проблем нет. Например, несмотря на то что email-адрес выглядит

[1] Мартин Фаулер. «Рефакторинг. Улучшение проекта существующего кода».

верным, при отправке сообщения может возникнуть ошибка. Или клиент предоставит положительное целое число, но оно не соответствует ни одному из значений идентификаторов в базе данных. Это означает, что ошибки все равно могут возникать при выполнении части кода, не затронутой проверками.

Если при выполнении метода после проверок предусловий что-то пойдет не так, нужно выдать другой тип исключения. Это не будет исключение, указывающее на недопустимый аргумент. Тип исключения должен указывать на то, что возникла ошибка, которую можно обнаружить только во время выполнения. Ошибку вызывает не сам метод, а некое внешнее условие, нарушающее ход выполнения метода. В листинге ниже показан пример.

Листинг 5.2. getRowById() выдает RuntimeException

```
public function getRowById(int id): array
{
    Assertion.greaterThan(id, 0, 'ID should be greater than 0');

    record = this.db.find(id);

    if (record == null) {
        throw new RuntimeException(
            'Не найдена запись с ID "{id}"'
        );
    }

    return record;
}
```

Здесь может выдаваться InvalidArgumentException

Это может привести к выдаче InvalidArgumentException или RuntimeException из кода, который вызывает базу данных

Это наш сценарий появления ошибки: не можем найти запись, поэтому выдаем RuntimeException

В конечном итоге у каждого вызываемого метода есть свои проверки предусловий, поэтому, помимо ошибок `RuntimeException`, возникающих в исходном коде, который делает вызовы к базе данных, мы можем столкнуться с `InvalidArgumentException` (или его родителем `LogicException`). Обычно мы просто позволяем этим исключениям «всплывать». Некоторые высокоуровневые механизмы обработки умеют обращаться с такими ошибками. Самое важное здесь — сценарии, которые сам метод распознает как ошибочные.

5.1.3. Счастливый путь

Счастливый путь («happy path») — это часть метода, где все идет хорошо и метод просто выполняет свою задачу. Если ваши методы небольшие, как и должно быть, вы увидите, что в этой части происходит немного событий. Зачастую бóльшая часть кода предназначен для обработки сценариев ошибок.

5.1.4. Проверки постусловий

Проверки постусловий можно добавить в метод для проверки того, что метод сделал именно то, что ожидалось. Можно анализировать значение перед возвратом или проверять состояние объекта перед его выдачей.

Листинг 5.3. someVeryComplicatedCalculation() осуществляет проверку постусловий

```
public function someVeryComplicatedCalculation(): int
{
    // ...
    result = /* ... */;
    Assertion.greaterThan(0, result);

    return result;
}
```

Эта проверка постусловий — просто проверка безопасности, иными словами, «такого не должно случиться»

На практике большинству методов не нужны проверки постусловий. Если вы пишете тесты для методов, вы уже *знаете*, что они возвращают правильные значения или что они правильно меняют состояние объекта на верное.

Если в вашем коде сильная типизация и вы не используете значения примитивного типа, то, задав параметры метода и возвращаемые значения с такими сильными типами, вы в результате получите целостный код, который не будет возвращать недопустимые значения. В конце концов, если возвращаемое значение — объект, то мы знаем, что он не сможет существовать в недопустимом состоянии.

Если вы работаете с унаследованным кодом с частыми случаями неявного приведения типов и без каких-либо проверок, то проверки постусловий могут пригодиться. Их можно использовать как проверки безопасности, чтобы убедиться, что в последующем коде не возникнет проблем.

СОЗДАВАЙТЕ НОВЫЕ МЕТОДЫ, ЧТОБЫ ОБОЙТИСЬ БЕЗ ИСПОЛЬЗОВАНИЯ ПРОВЕРОК ПОСТУСЛОВИЙ

Можно избавиться от проверок постусловий в методе аналогично тому, как мы делали с проверками предусловий, предоставляя значения примитивного типа для нужных объектов и возвращая такие же. Еще один вариант — обернуть метод с проверками постусловий в новый метод, в котором и будут проводиться проверки.

5.1.5. Возвращаемое значение

Метод должен что-то возвращать. Точнее, только методы-запросы. Мы обсудим эту тему подробнее в следующей главе.

Еще одно хорошее правило — это возвращать значения *сразу же*. Мы встречались с этим правилом при работе с исключениями. То же самое применимо и к возвращаемым значениям. Как только вы узнаете, что именно возвращать, возвращайте это немедленно, а не после прохождения через цепочку выражений с `if`.

5.2. НЕКОТОРЫЕ ПРАВИЛА ДЛЯ ИСКЛЮЧЕНИЙ

Мы видели, как исключения используются для проверок предусловий и постусловий, в том числе в сценариях ошибок. Рассмотрим правила проектирования для классов исключений.

5.2.1. Используйте собственные классы исключений только при необходимости

Добавление класса-исключения может оказаться очень полезным в некоторых ситуациях.

1. Если нужно поймать специфичный тип исключения выше:

```
try {
    // вероятно, выдаст исключение `SomeSpecific`
} catch (SomeSpecific exception) {
    // ...
}
```

2. Если существует несколько способов инстанцирования одного типа исключений:

```
final class CouldNotDeliverOrder extends RuntimeException
{
    public static function itWasAlreadyDelivered():
        CouldNotDeliverOrder
    {
        // ...
    }

    public static function insufficientQuantitiesInStock():
        CouldNotDeliverOrder
    {
        // ...
    }
}
```

3. Если нужно использовать именованные конструкторы для инстанцирования исключений:

```
final class CouldNotFindProduct extends RuntimeException
{
```

```
        public static function withId(
            ProductId productId
        ): CouldNotFindProduct {
            return new CouldNotFindProduct(
                'Не удалось найти продукт с ID "{productId}"'
            );
        }
    }

    throw CouldNotFindProduct.withId(/* ... *
```

Использование именованных конструкторов помогает сделать код на стороне клиента чище. Имя класса исключения в сочетании с именем метода-конструктора будет выглядеть, например, так: «Could not find product with ID...» («Не удалось найти продукт с ID...»). Сообщение собирается внутри класса исключения, а не с помощью внешнего вызова.

Наличие собственного класса исключения с подобным именованным конструктором дает возможность добавлять более одного именованного конструктора, что позволит использовать один и тот же класс исключения для выявления различных причин появления ошибки. Пример — в листинге ниже.

Листинг 5.4. Класс исключения с несколькими именованными конструкторами

```
final class CouldNotPersistObject extends RuntimeException
{
    public static function becauseDatabaseIsNotAvailable():
        CouldNotPersistObject
    {
        return new CouldNotPersistObject(/* ... */);
    }

    public static function becauseMappingConfigurationIsInvalid():
        CouldNotPersistObject
    {
        return new CouldNotPersistObject(/* ... */);
    }

    // ...
}
```

5.2.2. Именование недопустимых аргументов или классов логических исключений

Несмотря на распространенное мнение, имена классов исключений не обязаны содержать слово «Exception». Вместо этого существуют вспомогательные предложения, которые можно использовать для именования. Для обозначения недопустимых аргументов или логических ошибок используйте шаблон

«Invalid...», как, например, `InvalidEmailAddress`, `InvalidTargetPosition` или `InvalidStateTransition`.

5.2.3. Именование классов исключений времени исполнения

Для исключений времени исполнения подойдет завершение фразы «Sorry, [I] could not...» («К сожалению, не получилось...»). Слова в конце этого предложения станут именем вашего класса исключения. Это будут хорошие названия, так как они разъясняют, как система пыталась выполнить задачу, но у нее не получилось. Например, `CouldNotFindProduct`, `CouldNotStoreFile` или `CouldNotConnect`.

5.2.4. Используйте именованные конструкторы для указания причин ошибки

При использовании именованных конструкторов в их именах можно указывать условия возникновения исключения, как в листинге ниже.

Листинг 5.5. Именованный конструктор получает данные для использования

```
final class CouldNotFindStreetName extends RuntimeException
{
    public static function withPostalCode(
        PostalCode postalCode
    ): CouldNotFindStreetName {
        // ...
    }
}
```

В других случаях причину сбоя можно указать при помощи имени метода.

Листинг 5.6. Именованный конструктор указывает на причины ошибки

```
final class InvalidTargetPosition extends LogicException
{
    public static function becauseItIsOutsideTheMap(
        /* ... */
    ): InvalidTargetPosition {
        // ...
    }
}
```

5.2.5. Сопровождайте ошибки подробным описанием

Добавление именованных конструкторов удобно для клиентов, потому что сообщение об ошибке будет формировать сам конструктор исключения, а не клиент.

Листинг 5.7. Именованный конструктор формирует подробное сообщение

```
// Было:

final class CouldNotFindProduct extends RuntimeException
{
}

// В месте вызова:
throw new CouldNotFindProduct(
    'Не удалось найти продукт с ID "{productId}"'
);

// Стало:

final class CouldNotFindProduct extends RuntimeException
{
    public static function withId(
        ProductId productId
    ): CouldNotFindProduct {
        return new CouldNotFindProduct(
            'Не удалось найти продукт с ID "{productId}"'
        );
    }
}

// В месте вызова:
throw CouldNotFindProduct.withId(productId);
```

УПРАЖНЕНИЯ

1 Улучшите организацию выражений в следующем методе:

```
public function pop(): Element
{
    if (count(this.elements)) > 0) {
        lastElement = array_pop(this.elements);

        return lastElement;
    } else {
        throw new RuntimeException('Элементов больше нет');
    }
}
```

2 Какой тип исключения следует выдавать в случае ошибки «File not found» («Файл не найден»)?

а `RuntimeException` или его собственный подкласс.

б `InvalidArgumentException` или его собственный подкласс.

3 Какой тип исключения следует выдавать, если целое число, предоставленное клиентом, должно быть положительным, а оказалось отрицательным?

а `RuntimeException` или его собственный подкласс.

б `InvalidArgumentException` или его собственный подкласс.

ЗАКЛЮЧЕНИЕ

- Шаблоны для реализации методов нужны, чтобы начинать работу в подготовленной среде. Вы начинаете с анализа предоставленных аргументов, отбрасывая все недопустимые элементы при помощи выдачи исключений. Затем вы выполняете полезную работу, обрабатывая любые ошибки. И наконец, завершаете работу, после чего клиенту выдается возвращаемое значение.
- `InvalidArgumentException` должно указывать на проблему с аргументами, которые предоставил клиент. `RuntimeException` должно сообщать клиенту, что возникла проблема и что это не логическая ошибка.
- Создавайте собственные классы исключений и объявляйте именованные конструкторы, чтобы улучшить сообщения исключений и оптимизировать процессы их создания и выдачи.

ОТВЕТЫ К УПРАЖНЕНИЯМ

1 Вариант ответа:

```
public function pop(): Element
{
    if (count(this.elements)) == 0) {
        throw new RuntimeException('Элементов больше нет');
    }

    lastElement = array_pop(this.elements);

    return lastElement;
}
```

Переместите проверку ошибочных условий в начало метода

Всегда ищите возможность избавиться от else в if-выражении

2 Правильный ответ: **а**. Информация о том, что файл не найден, не читается из предоставленных аргументов, поэтому нужно выдавать `RuntimeException`.

3 Правильный ответ: **б**. Из содержимого предоставленных аргументов можно понять, что значение недопустимо, поэтому здесь нужно выдавать `InvalidArgumentException`.

6 Извлечение информации

В этой главе:

- ✓ Использование методов-запросов для извлечения информации
- ✓ Использование единого специфичного возвращаемого типа
- ✓ Проектирование объекта без раскрытия его внутренних данных
- ✓ Внедрение абстракций для вызовов-запросов
- ✓ Применение тестовых дублеров в вызовах-запросах

Объекты можно инстанцировать и иногда изменять. Объекты могут иметь методы для выполнения различных задач или для получения информации. В этой главе описывается, как реализовывать методы, которые возвращают информацию. В главе 7 мы рассмотрим методы, выполняющие задачи.

6.1. ИСПОЛЬЗУЙТЕ МЕТОДЫ-ЗАПРОСЫ ДЛЯ ИЗВЛЕЧЕНИЯ ИНФОРМАЦИИ

Выше мы вкратце рассмотрели командные методы. У этих методов возвращаемый тип — `void`, и они используются для получения побочных эффектов: изменения

состояния, отправки e-mail, сохранения файла и т. д. Командные методы не получают информацию. Если вам требуется получать информацию из объекта, используйте метод-запрос. У такого метода будет специфичный возвращаемый тип, и он не вызывает побочные эффекты.

Рассмотрим класс `Counter`.

Листинг 6.1. Класс Counter

```
final class Counter
{
    private int count = 0;

    public function increment(): void
    {
        this.count++;
    }
    public function currentCount(): int
    {
        return this.count;
    }
}

counter = new Counter();
counter.increment();

assertEquals(1, counter.currentCount());
```

Исходя из правил для методов-команд и методов-запросов, понятно, что `increment()` — это командный метод, так как он изменяет состояние объекта `Counter`. А `currentCount()` — это метод-запрос, так как он ничего не меняет, а просто возвращает текущее значение свойства `count`. При таком разделении вызов `currentCount()` в текущем состоянии объекта `Counter` всегда будет возвращать одно и то же значение.

Рассмотрим следующую альтернативную реализацию метода `increment()`.

Листинг 6.2. Альтернативная реализация метода increment()

```
public function increment(): int
{
    this.count++;

    return this.count;
}
```

Теперь метод вносит изменение и возвращает информацию. Это не очень хорошо для клиента, так как объект будет изменяться, даже если нам нужно только просмотреть его.

Лучше создавать методы двух категорий: безопасные методы, которые можно вызывать в любое время (и, по сути, любое количество раз), и другие «небезопасные» методы. Есть несколько способов этого добиться:

- Соблюдайте правило: метод должен быть либо командным, либо запросом. Это правило называют *принципом разделения методов-команд и методов-запросов* (command/query separation principle — CQS)[1]. Мы уже применяли его в начальной реализации объекта `Counter` (листинг 6.1): метод `increment()` был командным, а `currentCount()` — методом-запросом, и ни один не был одновременно командой и запросом.
- Создавайте объекты неизменяемыми (как мы говорили выше).

Если бы `Counter` был реализован как неизменяемый объект, то метод `increment()` был бы методом-модификатором, а его имя было бы более декларативным и звучало бы как `incremented()`.

Листинг 6.3. Альтернативная реализация объекта Counter

```
final class Counter
{
    private int count = 0;

    public function incremented(): Counter
    {
        copy = clone this;

        copy.count++;

        return copy;
    }

    public function currentCount(): int
    {
        return this.count;
    }
}

assertEquals(
    1,
    (new Counter()).incremented().currentCount()
);
assertEquals(
    2,
    (new Counter()).incremented().incremented().currentCount()
);
```

[1] Мартин Фаулер «CommandQuerySeparation» (2005), https://martinfowler.com/bliki/CommandQuerySeparation.html.

МЕТОД-МОДИФИКАТОР — ЭТО КОМАНДНЫЙ МЕТОД ИЛИ МЕТОД-ЗАПРОС?

Метод-модификатор не возвращает информацию. Он возвращает копию всего объекта, и после получения этой копии уже нельзя запросить о ней информацию, поэтому методы-модификаторы не являются методами-запросами. Но они также и не похожи на традиционные командные методы. Появление командного метода у неизменяемого объекта подразумевает изменение состояния объекта, чего быть не должно. И метод-модификатор создает новый объект, что напоминает метод-запрос.

И хотя это немного выходит за рамки концепции, можно сказать, что метод `incremented()` в листинге 6.3 отвечает на запрос «верни мне текущее значение, только увеличенное на 1».

УПРАЖНЕНИЕ

1 Какие методы похожи на методы-запросы?

- **а** `Name(): string`.
- **б** `changeEmailAddress(string emailAddress): void`.
- **в** `color(bool invert): Color`.
- **г** `findRecentMeetups(Date today): array`.

6.2. МЕТОДЫ-ЗАПРОСЫ ДОЛЖНЫ ИМЕТЬ ВОЗВРАЩАЕМЫЕ ЗНАЧЕНИЯ ЕДИНОГО ТИПА

Информация, возвращаемая методом, должна быть прогнозируемой. Не допускается использовать смешанные типы. В большинстве языков смешанные типы даже не поддерживаются, хотя PHP, динамически типизированный язык, является исключением. Например, в методе `isValid()` не указан возвращаемый тип, что позволяет возвращать несколько разных типов. Разработчиков это сбивает с толку.

Листинг 6.4. Метод isValid() легко может запутать

```
/**
 * @return string|bool
 */
public function isValid(string emailAddress)
{
    if (/* ... */) {
        return 'Invalid email address';
```

```
    }

    return true;
}
```

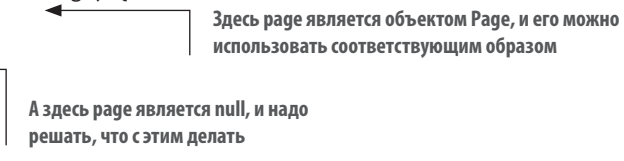
Если предоставленный email-адрес действителен, то `isValid()` возвращает true, в противном случае он вернет строку. Правильнее будет обеспечить возврат значений одного типа.

Здесь уместно обсудить еще одну ситуацию. Взгляните на следующий метод, который не предполагает нескольких возвращаемых типов (только один тип — объект `Page`), но он может возвращать и `null`:

```
public function findOneBy(type): Page?
{
}
```

Вызывающему метод всегда приходится проверять, является ли возвращаемое значение объектом `Page` или это `null`.

```
if (page instanceof Page) {
    // ...
} else {
    // ...
}
```



Возвращение `null` не всегда проблема. Но следует убедиться, что клиенты метода учитывают такую ситуацию. В PHP существуют инструменты для статического анализа, такие как PHPStan (github.com/phpstan/phpstan) и Psalm (github.com/vimeo/psalm), способные проверять условие возвращения `null`. Кроме того, на риски возможного исключения по null pointer может указывать IDE. В Java есть CheckerFramework (checkerframework.org), который во время компиляции выводит уведомления, что клиенты не обрабатывают возможное `null`-значение.

В большинстве случаев лучше рассмотреть альтернативы возвращению `null`. Например, в следующем методе, `getById()`, который должен получать сущность `User` по ID пользователя, не возвращается `null`, если пользователь не найден, а выдается исключение. В конце концов, клиент ожидает, что пользователь существует, он ведь даже предоставляет его ID. Ему не нужен `null` в ответ.

Листинг 6.5. getById() возвращает User или выдает исключение

```
public function getById(id): User
{
    user = /* ... */;

    if (!user instanceof User) {
```

```
        throw UserNotFound.withId(id);
    }

    return user;
}
```

Еще один вариант — возвращать объект, который будет представлять случай `null`. Такой объект называют `null`-объектом. Клиентам не придется проверять ответ на `null`, так как у возвращаемого объекта уже будет верный тип, как указано в сигнатуре метода.

Листинг 6.6. Возвращайте null-объект, если нужно

```
public function findOneByType(PageType type): Page
{
    page = /* ... */;  <-- Попытка найти страницу

    if (!page instanceof Page) {
        return new EmptyPage();
    }

    return page;
}
```

УКАЗЫВАЙТЕ НА НЕОПРЕДЕЛЕННОСТЬ В НАЗВАНИИ МЕТОДА

Имя метода может выражать неопределенность в отношении того, вернет ли метод значение ожидаемого типа. В предыдущих примерах мы использовали `getIserId()` вместо, например, `findUserId()`, чтобы сообщить клиенту, что метод получит (get) объект `User`, а не просто попытается найти его и, возможно, вернется пустым.

Еще одной альтернативой возвращению `null` является возвращение результата в виде сущности того же типа, но без содержимого. Если метод должен найти и вернуть количество объектов в массиве, возвращайте пустой массив, когда ничего не найдено.

Листинг 6.7. Вместо null возвращайте пустой массив

```
public function eventListenersForEvent(string eventName): array
{
    if (!isset(this.listeners[eventName])) {
        return [];
    }

    return this.listeners[eventName];
}
```

У других типов будут другие пустые возвращаемые значения. Например, если возвращаемый тип — `int`, то пустое значение может быть `0` (или `1`); в случае строк — `' '` (или `'N/A'`).

Если вы обнаружили, что используете существующий метод, в котором несколько возвращаемых типов, или он возвращает `null` вместо чего-то более надежного, то, возможно, стоит написать новый метод, который расставит все по местам. В следующем примере существующий метод `findOneByType()` возвращает объект `Page` или `null`. Если мы хотим избавить клиента от обработки ситуаций с `null` и просто получать объект `Page`, можно обернуть вызов `findByType()` в новый метод под названием `getOneByType()`.

Листинг 6.8. Метод getOneByType() оборачивает метод findOneByType(), который может возвращать null

```
public function getOneByType(PageType type): Page
{
    page = this.findOneByType(type);

    if (!page instanceof Page) {
        throw PageNotFound.withType(type);  ◄— Вместо возврата null выдается исключение
    }

    return page;
}
```

УПРАЖНЕНИЕ

2 Какие утверждения верны для методов-запросов?

- **а** Они должны вызывать видимый побочный эффект.
- **б** Они не должны возвращать смешанные типы (например, `bool|int`).
- **в** Иногда имеет смысл возвращать из них `null`.
- **г** Иногда имеет смысл ничего из них не возвращать (`void`).

6.3. ИЗБЕГАЙТЕ ИСПОЛЬЗОВАНИЯ МЕТОДОВ-ЗАПРОСОВ, РАСКРЫВАЮЩИХ ВНУТРЕННИЕ ДАННЫЕ ОБЪЕКТОВ

Самая простая реализация метода-запроса —непосредственное возвращение свойства объекта. Такие методы называют *геттерами*, и они позволяют клиентам получать (get) внутренние данные объекта.

«НО, СОГЛАСНО КОНВЕНЦИЯМ JAVABEANS, У КАЖДОГО СВОЙСТВА ДОЛЖЕН БЫТЬ ГЕТТЕР!»

Это правда, конвенции JavaBeans предписывают создавать для объектов конструктор без аргументов и объявлять геттеры и сеттеры для каждого свойства (http://mng.bz/1woy). Как вы, возможно, догадались после прочтения предыдущих глав, каждое из правил или рекомендаций в этой книге расходится с данной конвенцией. Наличие конструктора без аргументов приведет к тому, что объекты будут создаваться с изначально недопустимым состоянием. Вызов каждого сеттера по отдельности может привести к таким же недопустимым промежуточным состояниям объекта. А если внутренние данные объекта выражены явно, то становится сложно что-то в них изменить, не нарушая работу клиентов. Единственный тип объектов, который можно спроектировать, соблюдая правила JavaBeans, — это объект для передачи данных (мы обсуждали их в разделах 3.13 и 4.3).

Клиентам обычно нужно получать данные, чтобы производить с ними вычисления или основывать на них решения. Так как внутренние данные объектов лучше не раскрывать, то необходимо следить, чтобы возвращаемые простыми геттерами значения должным образом использовались клиентами. Но многое, что клиент может делать с информацией в объекте, способен выполнять и сам объект.

Первый пример — это метод `getItems()`, который возвращает товары в корзине, чтобы клиент мог их подсчитать. Вместо того чтобы раскрывать сами товары, корзина может содержать метод для подсчета товаров.

Листинг 6.9. Альтернатива для подсчета товаров

```
// Было:

final class ShoppingBasket
{
    // ...

    public function getItems(): array
    {
        return this.items;
    }
}

count(basket.getItems());

// Стало

final class ShoppingBasket
{
    // ...
```

```
    public function itemCount(): int
    {
        return count(this.items);
    }
}

basket.itemCount();
```

Именование методов-запросов тоже важная тема. Мы не использовали название `getItemCount()` или `countItems()`, так как они звучат как команды, которые приказывают объекту что-либо сделать. Вместо этого мы назвали метод `itemCount()`, чтобы рассматривать подсчет товаров как свойство корзины, о котором можно получать информацию.

ЧТО ДЕЛАТЬ С НЕОДНОЗНАЧНЫМИ НАЗВАНИЯМИ?

Что, если у объекта есть свойство `name`? Геттер для этого свойства будет называться `name()`, но слово name может быть и глаголом. Мы уже встречались с подобной ситуацией, когда использовали слово count, которое тоже может быть глаголом.

И хотя значение слов всегда будет предметом споров, в большинстве случаев неоднозначные ситуации можно разрешать, определяя правильный контекст. Как только вы уловите разницу между командными методами и методами-запросами, вам будет проще замечать, когда метод должен возвращать информацию (например, возвращать значение свойства `name`) или изменять состояние объекта (например, изменять значение свойства `name` и ничего не возвращать). Это даст читателю важную подсказку, следует ли интерпретировать слово как глагол (что чаще всего указывает на командный метод) или как существительное (которое чаще всего является признаком метода запроса).

Переписывая таким образом классы, вы сможете добавлять в объект больше логики, сохраняя сведения о предмете в пределах самого класса, а не распределять их по всей кодовой базе.

Вот еще пример: клиенты могут вызвать на объекте метод-запрос, а затем другой метод, который использует значение, возвращаемое первым методом.

Листинг 6.10. Клиенты используют геттеры класса Product для принятия решений

```
final class Product
{
    public function shouldDiscountPercentageBeApplied(): bool
    {
        // ...
    }
```

Продукт включает параметр, определяющий, следует ли применять к нему процентную скидку. Если такая скидка неприменима, то есть возможность использовать фиксированную скидку

```
    public function discountPercentage(): Percentage
    {
        // ...
    }

    public function fixedDiscountAmount(): Money
    {

    }
}

amount = new Money(/* ... */);
if (product.shouldDiscountPercentageBeApplied()) {
    netAmount = product.discountPercentage().applyTo(amount);
} else {
    netAmount = amount.subtract(product.fixedDiscountAmount());
}
```

Клиенты продукта могут вычислить стоимость товара при помощи вызова метода applyDiscountPercentage() и использовать его возвращаемое значение для применения процентной или фиксированной скидки

Для хранения сведений о вычислении скидки для конкретного товара можно использовать метод с соответствующим названием: `calculateNetAmount()`.

Листинг 6.11. calculateNetAmount() предлагает альтернативу

```
final class Product
{
    public function calculateNetAmount(Money amount): Money
    {
        if (this.shouldDiscountPercentageBeApplied()) {
            return this.discountPercentage().applyTo(amount);
        }

        return amount.subtract(this.fixedDiscountAmount());
    }

    private function shouldDiscountPercentageBeApplied(): bool
    {
        // ...
    }

    private function discountPercentage(): Percentage
    {
        // ...
    }

    private function fixedDiscountAmount(): Money
    {
    }
}

amount = new Money(/* ... */);
netAmount = product.calculateNetAmount(amount);
```

Эти методы могут остаться приватными, или же возможно использовать те свойства, которые они раскрывают

Эти методы могут остаться приватными, или же возможно использовать те свойства, которые они раскрывают

Помимо отсутствия необходимости повторять эту логику в разных вызовах, вы получаете еще два преимущества. Первое: можно не раскрывать внутренние данные, такие как данные о скидке. Второе: схемы вычислений можно менять и тестировать в одном месте.

Короче говоря, всегда избегайте создания методов-запросов, которые будут раскрывать внутренние данные объекта:

- Создавайте эффективные методы, адаптируйтесь к потребностям клиентов.
- Переместите вызов метода в объект, это позволит объекту самому решать, что делать.

Таким образом вы защитите внутреннюю информацию объекта, и клиенты будут использовать только его публичные интерфейсы (рис. 6.1).

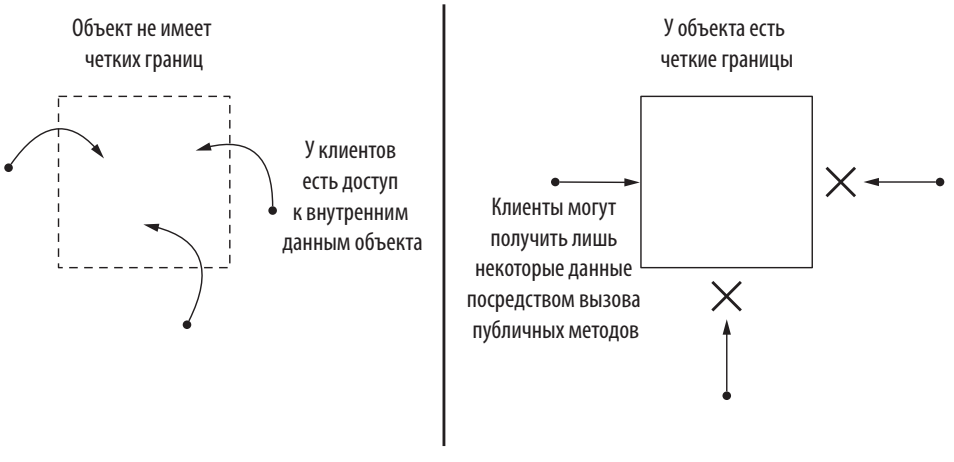


Рис. 6.1. Можно представить, что у объекта есть границы. Вместо того чтобы позволять клиентам пересекать эти границы для извлечения информации, лучше явно указать, какие данные и какое поведение должны быть доступны клиентам

ПРАВИЛА ИМЕНОВАНИЯ ГЕТТЕРОВ

Как вы, возможно, заметили, я не использовал традиционные префиксы get для именования геттеров, как, например, в `discountPercentage()` (листинг 6.11). Это название показывает, что метод не является командным, он просто предоставляет информацию. Название метода — это описание части нужной информации, а не инструкция для объекта о том, как ее получить.

УПРАЖНЕНИЕ

3 Взгляните, как следующие классы, `Order` и `Line`, позволяют клиентам получать всю необходимую информацию для вычисления итоговой стоимости заказа.

```
final class Line
{
    private int quantity;
    private Money tariff;

    // ...

    public function quantity(): int
    {
        return this.quantity;
    }

    public function tariff(): Money
    {
        return this.tariff;
    }
}

final class Order
{
/**
 * @var Line[]
 */
    private array lines = [];

    // ...

    /**
     * @return Line[]
     */
    public function lines(): array
    {
        return this.lines;
    }
}

totalAmount = new Money(0);
foreach (order.lines() as line) {
    totalAmount = totalAmount.add(
        new Money(
            line.quantity() * line.tariff()
        )
    );
}
```

Перепишите классы `Order` и `Line` так, чтобы объекту `Order` больше не приходилось раскрывать внутренние данные массива `lines`, какой-либо из объектов `Line`, перечисленных в массиве, или их свойства `tariff` и `quantity`.

6.4. ЗАДАВАЙТЕ СПЕЦИФИЧНЫЕ МЕТОДЫ И ВОЗВРАЩАЕМЫЕ ТИПЫ ДЛЯ НЕОБХОДИМЫХ ЗАПРОСОВ

Когда вам нужна конкретная информация, убедитесь, что вы сформулировали конкретный вопрос и знаете, как должен выглядеть ответ. Например, если вы работаете с кодом и вам нужен сегодняшний курс обмена USD к EUR, то, чтобы получить эту информацию, следует вызвать некоторые веб-сервисы, например https://fixer.io/. Возможно, вы сразу напишете для этого код.

Листинг 6.12. Класс CurrencyConverter

```
final class CurrencyConverter
{
    public function convert(Money money, Currency to): Money
    {
        httpClient = new CurlHttpClient();
        response = httpClient.get(
            'http://data.fixer.io/api/latest?access_key=...' .
                '&base=' . money.currency().asString() .
                '&symbols=' . to.asString()
        );
        decoded = json_decode(response.getBody());
        rate = (float)decoded.rates[to.asString()];

        return money.convert(to, rate);
    }
}
```

Этот небольшой фрагмент кода содержит много ошибок (и я не говорю об отказах сети, ошибках ответа, недействительных JSON, модифицированной структуре ответа и о том, что `float` — не самый надежный тип данных для работы с денежными единицами). В нем можно обнаружить и концептуальные ошибки. Все, что нам нужно от этого кода в данный момент, — это ответ на вопрос: каков текущий курс обмена USD к EUR?

Исправленный код, отвечающий на этот вопрос, содержит два новых класса: `FixedApi` и `ExchangeRage`. У первого класса есть единственный метод — `exchangeRateFor()`, представляющий вопрос, который желает задать `CurrencyConversion`. Второй класс — `ExchangeRate` — это ответ.

Листинг 6.13. Классы FixedApi и ExchangeRate

```
final class FixerApi
{
    public function exchangeRateFor(
        Currency from,
        Currency to
    ): ExchangeRate {
        httpClient = new CurlHttpClient();
        response = httpClient.get(/* ... */);
        decoded = json_decode(response.getBody());
        rate = (float)decoded.rates[to.asString()];

        return ExchangeRate.from(from, to, rate);
    }
}

final class ExchangeRate
{
    public static function from(
        Currency from,
        Currency to,
        float rate
    ): ExchangeRate {
        // ...
    }
}

final class CurrencyConverter
{
    private FixerApi fixerApi;

    public function __construct(FixerApi fixerApi)
    {
        this.fixerApi = fixerApi;
    }

    public function convert(Money money, Currency to): Money
    {
        exchangeRate = this.fixerApi
            .exchangeRateFor(
                money.currency(),
                to
            );

        return money.convert(exchangeRate);
    }
}
```

Мы создаем метод FixedApi.exchangeRateFor(), который будет представлять собой задаваемый вопрос: каков текущий обменный курс … к …?

Новый класс будет представлять ответ на вопрос

CurrenceConverter будет получать внедренный экземпляр FixedApi, чтобы при необходимости узнать текущий курс обмена

Класс «ответа» `ExchangeRate` должен быть максимально полезен для клиента. Этот класс потенциально может быть использован и при других обстоятельствах, но это не обязательно.

Здесь важно, что создание метода `exchangeRateFor()` со специфичным возвращаемым типом улучшает обмен информацией в коде. При чтении кода метода `convert()` явно видно, что метод нуждается в информации, задается вопрос, возвращается ответ, который затем используется для выполнения операций. Заметьте, что мы лишь провели рефакторинг кода; его поведение осталось прежним.

6.5. ЗАДАВАЙТЕ АБСТРАКЦИЮ ДЛЯ ЗАПРОСОВ, КОТОРЫЕ ВЫХОДЯТ ЗА ГРАНИЦЫ СИСТЕМЫ

На вопрос «Каков текущий обменный курс?» из предыдущего раздела приложение своими силами ответить не может, так как эта информация не находится в памяти приложения. Чтобы ответить на этот вопрос, приложению необходимо выйти за свои границы. В данном случае необходимо установить соединение с удаленным сервисом, доступным по сети. Другие примеры выхода приложения за границы — когда приложение обращается к файловой системе, чтобы загрузить или сохранить файл, или когда для получения текущего времени используется системное время.

Когда приложение выходит за свои границы, необходимо создавать абстракцию, которая позволит скрыть внутренние низкоуровневые подробности вызовов методов.

Абстракция здесь имеет два значения, и она будет работать при наличии двух условий:

- Использование сервисного интерфейса вместо сервисного класса.
- Хранение деталей реализации закрытыми.

С правильной абстракцией вы сможете запускать код в тестовом сценарии, не совершая вызовов к сети или файловой системе. Это также позволит переключать способы реализации, не меняя клиентский код, а лишь переписав реализацию сервисного интерфейса.

Сначала мы обсудим неудачный пример реализации абстракции. Взглянем еще раз на класс `FixedApi`. В нем сетевой вызов осуществляется непосредственно при помощи класса `CurlHttpClient`.

Листинг 6.14. Использование экземпляра CurlHttpClient для соединения с API

```
final class FixerApi
{
    public function exchangeRateFor(
        Currency from,
        Currency to
```

```
    ): ExchangeRate {
        httpClient = new CurlHttpClient();
        response = httpClient.get(/* ... */);
        decoded = json_decode(response.getBody());
        rate = (float)decoded.rates[to.asString()];

        return ExchangeRate.from(from, to, rate);
    }
}
```

Вместо того чтобы инстанцировать и использовать этот класс, можно задать для него интерфейс и передавать его экземпляр классу `FixedApi`, как в примере ниже.

Листинг 6.15. Добавление интерфейса HttpClient и его использование в классе FixerApi

```
interface HttpClient                                    <-- Сначала создаем интерфейс для HTTP-клиентов
{
    public function get(url): Response;
}

final class CurlHttpClient implements HttpClient        <-- Проверяем, что существующий CurlHttpClient использует новый интерфейс HttpClient
{
    // ...
}

final class FixerApi
{
    public function __construct(HttpClient httpClient)  <-- Передаем интерфейс, а не конкретный класс
    {
        this.httpClient = httpClient;
    }

    public function exchangeRateFor(
        Currency from,
        Currency to
    ): ExchangeRate {
        response = this.httpClient.get(/* ... */);      <-- Необходимо изменить код, чтобы использовать новый интерфейс и его метод get()
        decoded = json_decode(response.getBody());
        rate = (float)decoded.rates[to.asString()];

        return ExchangeRate.from(from, to, rate);
    }
}
```

Теперь можно менять реализацию интерфейса `HttpClient`, поскольку мы полагаемся на интерфейс, а не на конкретную реализацию. Это будет полезно, если когда-нибудь потребуется перейти на другую реализацию `HttpClient`. Но мы все еще не создали основную абстракцию. Что, если потребуется переключиться на другой API? Весьма вероятно, что новый API не будет присылать такой же

JSON-ответ. Или нам захочется создать свою локальную базу данных с обменными курсами. В таком случае нам больше не нужен будет HTTP-клиент.

Чтобы избавиться от низкоуровневых деталей реализации, необходимо выбрать более абстрактное имя, соответствующее целям. Нам нужно средство получения обменного курса. Откуда его получать? Оттуда, где он есть. Или оттуда, где им управляет некая «коллекция». Хорошим названием может быть `ExchangeRateProvider` или просто `ExchangeRate`, если рассматривать этот сервис в качестве коллекции известных значений обменных курсов. В листинге ниже показано, как это может выглядеть.

Листинг 6.16. Внедрение абстрактного сервиса ExchangeRate

```
/**
 * Извлечь метод вопроса и сделать его публичным
 * в абстрактном сервисе `ExchangeRates`:
 */
interface ExchangeRates
{
    public function exchangeRateFor(
        Currency from,
        Currency to
    ): ExchangeRate;
}

final class FixerApi implements ExchangeRates
{
    private HttpClient httpClient;

    public function __construct(HttpClient httpClient)
    {
        this.httpClient = httpClient;
    }

    public function exchangeRateFor(
        Currency from,
        Currency to
    ): ExchangeRate {
        response = this.httpClient.get(/* ... */);
        decoded = json_decode(response.getBody());
        rate = (float)decoded.data.rate;

        return ExchangeRate.from(from, to, rate);
    }
}

final class CurrencyConverter
{
    private ExchangeRates exchangeRates;

    public function __construct(ExchangeRates exchangeRates)
    {
```

Существующий класс FixerApi должен использовать новый интерфейс ExchangeRates

Вместо объекта Fixer теперь можно использовать экземпляр ExchangeRates

```
        this.exchangeRates = exchangeRates;
    }

    // ...

    private function exchangeRateFor(
        Currency from,
        Currency to
    ): ExchangeRate {
        return this.exchangeRates.exchangeRateFor(from, to);
    }
}
```

Используем новый сервис для получения ответа на вопрос

И последнее улучшение — можно направить все существующие вызовы к приватному методу `exchangeRatesFor()`, это просто посредник для сервиса `ExchangeRates`.

Создав интерфейс для существующего класса, мы сделали первый шаг к хорошей абстракции. Спрятав все детали реализации в интерфейсе, мы сделали второй шаг. Итак, у нас теперь есть надлежащая абстракция для получения курсов обмена валют. Это дает два преимущества:

- Можно легко перейти к другому поставщику обменных курсов. Новый класс правильно использует интерфейс `ExchangeRates`, поэтому класс `CurrencyConverter` не нужно будет изменять, поскольку он зависит от абстракции `ExchangeRates`.
- Можно написать модульный тест для класса `CurrencyConverter` и внедрить тестовый дублер для `ExchangeRates`, который не осуществляет сетевых вызовов. Это сделает тест быстрым и надежным.

Кстати, если вы знакомы с принципами SOLID, то уже встречались с простым правилом абстракции сервисных зависимостей, известным как *принцип инверсии зависимостей*. Подробнее о нем можно прочесть в книгах и статьях Роберта Мартина (Robert C. Martin)[1].

НЕ ДЛЯ КАЖДОГО ВОПРОСА НУЖНО СОЗДАВАТЬ ОТДЕЛЬНЫЙ СЕРВИС

В предыдущих примерах для вопроса «Каков текущий обменный курс?» требовалось создавать отдельный сервис. Это вопрос, на который приложение своими силами ответить не может. В большинстве случаев не всегда нужно создавать новый объект. Рассмотрим следующие варианты:

1. Улучшить именование переменных, чтобы сделать взаимодействие в коде более понятным.

[1] Например: Robert C. Martin. «The Dependency Inversion Principle», https;//mng.bz/9woa. Другие статьи о принципах SOLID можно почитать здесь: https://mng.bz/j50y.

2. Выделить приватный метод, который представляет собой вопрос, и ответ на него (как мы делали до этого, переместив логику в приватный метод `exchangeRateFor()`).

Если метод разрастается, требует отдельного тестирования или выходит за пределы приложения, необходимо создавать отдельный класс. Это поможет ограничить количество используемых типов объектов и сделает код более удобочитаемым, ведь не придется пересматривать множество классов, чтобы выяснить, как все работает.

УПРАЖНЕНИЕ

4 Какие две вещи нужно сделать, чтобы создать абстракцию для сервиса?

- **а** Создать абстрактный класс для сервиса.
- **б** Создать интерфейс для сервиса.
- **в** Использовать высокоуровневое именование, которое позволит хранить низкоуровневые детали реализации скрытыми.
- **г** Создать как минимум две реализации абстрактного сервиса.

6.6. ИСПОЛЬЗУЙТЕ ЗАГЛУШКИ В ТЕСТОВЫХ ДУБЛЕРАХ ДЛЯ МЕТОДОВ-ЗАПРОСОВ

Когда вы внедряете абстракцию для запросов, вы создаете полезную точку расширения. Вы можете менять реализацию способа получения ответа на вопрос. Тестировать такую логику также будет легче. Вместо того чтобы тестировать сервис `CurrencyConverter`, только когда доступно интернет-соединение (и удаленный сервис), теперь можно тестировать логику, заменив `ExchangeRates` сервисом, в котором уже имеются необходимые и предсказуемые ответы.

Листинг 6.17. Тестирование CurrencyConverter при помощи ExchangeRatesFake

```
final class ExchangeRatesFake implements ExchangeRates
{
    private array rates = [];

    public function __construct(
        Currency from,
        Currency to,
        float rate
    ) {
        this.rates[from.asString()][to.asString()] =
        ExchangeRate.from(from, to, rate);
    }
```

Это фиктивная реализация сервиса ExchangeRates, которую можно настроить на возврат любого значения обменного курса

```
    public function exchangeRateFor(
        Currency from,
        Currency to
    ): ExchangeRate {
        if (!isset(this.rates[from.asString()][to.asString()])) {
            throw new RuntimeException(
                'Не удалось определить обменный курс [...] к [...]
            );
        }
        return this.rates[from.asString()][to.asString()];
    }
}

/**
 * @test
 */
public function it_converts_an_amount_using_the_exchange_rate(): void
{
    exchangeRates = new ExchangeRatesFake();
    exchangeRates.setExchangeRate(
        new Currency('USD'),
        new Currency('EUR'),
        0.8
    );

    currencyConverter = new CurrencyConverter(exchangeRates);

    converted = currencyConverter
        .convert(new Money(1000, new Currency('USD')));

    assertEquals(new Money(800, new Currency('EUR')), converted);т
}
```

Можно использовать этот фиктивный сервис в модульном тесте для CurrencyConverter

Создание фиктивного сервиса ExchangeRates

Передача фиктивного сервиса в CurrencyConverter

В этом тесте мы сосредоточились на логике метода `convert()`, вместо того чтобы устанавливать сетевое соединение, проводить парсинг ответа JSON и т. п. Это делает тест детерминированным и стабильным.

ИМЕНОВАНИЕ ТЕСТОВЫХ МЕТОДОВ

В листинге 6.17 имена тестовых методов были составлены в так называемом змеином регистре: с использованием символов нижнего регистра и символа нижнего подчеркивания в качестве разделителя слов. Если следовать стандарту именования методов, то его имя будет выглядеть так: `itConvertsAnAmountUsingTheExchangeRate()`. Большинство стандартов также предлагают использовать относительно короткие имена, но вариант `it_converts_an_amount_using_the_exchange_rate()` точно не назвать коротким. Так как основная цель тестовых методов отличается от цели обычных методов, то и принципы именования этих методов должны быть иными:

1. Имена тестовых методов описывают поведение объекта. Самое лучшее описание — это полное предложение.
2. Так как имена тестовых методов должны представлять собой предложения, их длина будет превышать длину имен обычных методов.

Но они все равно должны быть удобочитаемыми (поэтому лучше использовать змеиный регистр). Если вы не привыкли к таким правилам, легче всего начинать имена тестовых методов с `it_`. Это позволит настроиться на описание поведения конкретного объекта. И хотя это уже само по себе хорошее начало, вы заметите, что не каждое название тестового метода стоит начинать с `it_`. Например, можно использовать `when_` и `if_`.

Фиктивный объект (*fake*) — это только один из видов тестовых дублеров. Его можно охарактеризовать как усложненное до определенной степени поведение, как в настоящей реализации, которая будет использоваться в продакшене. При тестировании также можно использовать *заглушки* (*stub*), которые будут заменять настоящие сервисы. Заглушка — это тестовый дублер, который возвращает фиксированные значения. Когда бы мы ни вызвали метод `exchangeRateFor()`, он всегда будет возвращать одно и то же значение, как в примере ниже.

Листинг 6.18. ExchangeRateStub всегда возвращает одно и то же значение

```
final class ExchangeRatesStub ◄── Это пример реализации заглушки ExchangeRates
{
    public function exchangeRateFor(
        Currency from,
        Currency to
    ): ExchangeRate {
        return ExchangeRate.from(from, to, 1.2); ◄── Возвращаемое значение жестко закодировано

    }
}
```

Важной характеристикой фиктивных объектов и заглушек служит то, что в тестовом сценарии предположения о количестве или порядке вызовов методов недопустимы. Исходя из природы методов-запросов, стоит иметь в виду, что их вызов не должен сопровождаться побочными эффектами, поэтому их можно вызывать многократно или вообще не вызывать. Создание утверждений о вызовах методов-запросов приводит к тому, что тесты не соблюдают достаточной дистанции с реализацией тестируемых классов.

Для командных методов ситуация противоположная: нужно проверить факт вызова, количество вызовов и, возможно, их порядок. Мы вернемся к этому в следующей главе.

НЕ ИСПОЛЬЗУЙТЕ MOCK-ИНСТРУМЕНТЫ ДЛЯ СОЗДАНИЯ ФИКТИВНЫХ ОБЪЕКТОВ И ЗАГЛУШЕК

Mock-фреймворки часто используются для быстрого создания тестовых дублеров. Я не рекомендую применять эти фреймворки для создания фиктивных объектов и заглушек. Код будет короче, но его станет тяжело читать и поддерживать.

Даже если вы предпочитаете пользоваться такими инструментами, я рекомендую применять их только для создания *пустышек (dummy)* (то есть тестовых дублеров, которые не возвращают ничего значимого и используются только для того, чтобы передавать их в качестве неиспользуемых аргументов). В случае с заглушками и фиктивными объектами mock-инструменты могут даже помешать проектированию. Они будут проверять, вызывался ли метод и сколько раз, но они усложнят рефакторинг, так как названия методов зачастую будут представлены строками и инструмент для рефакторинга может не распознать их как реальные имена методов.

Не забудьте протестировать настоящую реализацию, которая использует HTTP-соединение для получения обменного курса. Нужно убедиться, что все работает хорошо. Но теперь нам уже не нужно беспокоиться о тестировании логики конвертации как таковой, а только от том, как реализация сообщается с внешними сервисами.

Листинг 6.19. Тест для FixerApi — интеграционный

```
/**
 * @test
 */
public function it_retrieves_the_current_exchange_rate(): void
{
    exchangeRates = new FixerApi(new CurlHttpClient());

    exchangeRate = exchangeRates.exchangeRateFor(
        new Currency('USD'),
        new Currency('EUR')
    );

    // Проверить результат здесь...
}
```

Позже вы поймете, что нужно приложить еще некоторые усилия, чтобы сделать тест более стабильным. Возможно, придется запустить собственный сервер для обменных курсов, который будет копией настоящего. Или же будет возможность использовать среду-песочницу, предоставляемую службой поддержи настоящего сервиса.

Заметьте, что тогда тест уже не будет считаться модульным: он не проверяет поведение объекта, размещенного в памяти. Этот тест можно назвать интеграционным, так как он проверяет интеграцию объекта с внешним сервисом.

6.7. МЕТОДЫ-ЗАПРОСЫ ДОЛЖНЫ ИСПОЛЬЗОВАТЬ ДРУГИЕ МЕТОДЫ-ЗАПРОСЫ, А НЕ КОМАНДНЫЕ МЕТОДЫ

Как мы уже обсуждали, у командных методов есть побочные эффекты. Командные методы что-то изменяют, что-то сохраняют, отправляют письма и т. д. Методы-запросы, в свою очередь, ничего подобного не делают. Они просто возвращают информацию. Обычно методу-запросу нужно взаимодействовать с другими объектами, которые помогают сформулировать ответ. Если в коде мы разделяем методы на командные и запросы, то цепочка вызовов, которая начинается с запроса, не должна содержать командных методов. Запросы не должны создавать побочные эффекты, а командные методы в цепочке будут нарушать это правило.

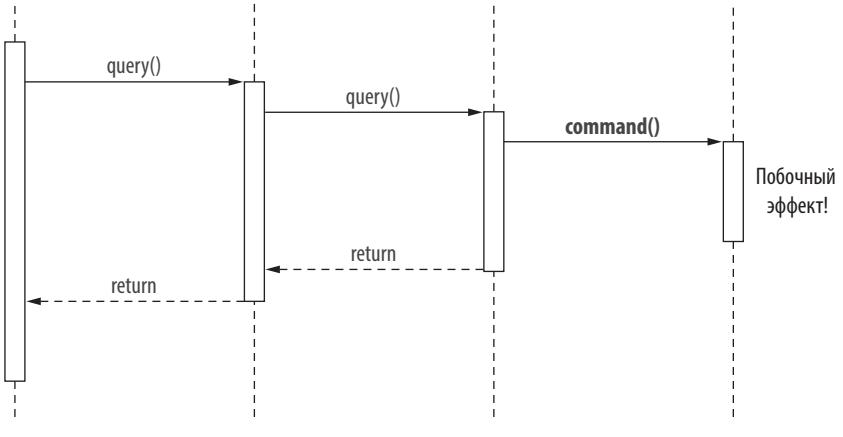


Рис. 6.2. В этой цепочке вызовов не должно быть командных методов

Существует несколько исключений из этого правила. Рассмотрим метод-контроллер для веб-приложения, который будет вызываться для регистрации нового пользователя. У этого метода есть побочный эффект: где-то в цепочке команд новый пользователь будет сохраняться в базе данных. В такой ситуации мы использовали бы возвращаемый тип `void` для самого контроллера, но веб-приложения должны всегда возвращать HTTP-ответ. Поэтому контроллеру придется возвращать хотя бы что-то.

Листинг 6.20. Контроллер всегда будет что-то возвращать

```
final class RegisterUserController
{
    private RegisterUser registerUser;

    public function __construct(
        RegisterUser registerUser
    ) {
        this.registerUser = registerUser;
    }

    public function execute(Request request): Response
    {
        newUser = this.registerUser
            .register(request.get('username'));

        return new Response(200, json_encode(newUser));
    }
}
```

Технически контроллер будет нарушать принцип разделения команд/запросов, но эту ситуацию не обойти. По крайней мере, мы всегда должны возвращать пустой ответ `200 OK` или нечто подобное. Но это не принесет пользы тем, кто работает с клиентской частью, ведь им нужно будет делать `POST`-запросы на регистрацию пользователя и получать ответ с JSON-структурой, включающей информацию о новом пользователе.

Чтобы найти решение, нужно разделить действия контроллера на две части: регистрацию нового пользователя и возвращение ответа. Лучше всего наделить пользователя идентификатором до того, как будет вызван сервис `RegisterUser`, чтобы сервису не приходилось ничего возвращать и его получилось бы сделать настоящим командным методом. Это показано в листинге ниже.

Листинг 6.21. Контроллер можно разделить на две части: команду и запрос

```
final class RegisterUserController
{
    private UserRepository userRepository;
    private RegisterUser registerUser;
    private UserReadModelRepository userReadModelRepository;

    public function __construct(
        UserRepository userRepository,
        RegisterUser registerUser,
        UserReadModelRepository userReadModelRepository
    ) {
        this.userRepository = userRepository;
        this.registerUser = registerUser;
        this.userReadModelRepository = userReadModelRepository;
    }
```

```
    public function execute(Request request): Response
    {
        userId = this.userRepository.nextIdentifier();

        this.registerUser
            .register(userId, request.get('username'));

        newUser = this.userReadModelRepository.getById(userId);

        return new Response(200, json_encode(newUser));
    }
}
```

register() — командный метод

getById() — метод-запрос

ИНОГДА ПРИНЦИП РАЗДЕЛЕНИЯ НА ЗАПРОС И КОМАНДУ НЕ ИМЕЕТ СМЫСЛА

Такое разделение необходимо использовать почти всегда, но оно не является непреложной истиной. По большому счету в программировании не должно быть подобных правил.

Чаще всего препятствием для использования этого принципа становится многопоточность. Пример такой ситуации — метод `nextIdentity()` ниже. В нем генерируется уникальный идентификатор для сущности, которую требуется сохранить. Идентификаторами служат числа в последовательности 1, 2, 3 и т. д.

```
final class EntityRepository
{
    public function nextIdentity(): int
    {
        // ...
    }
}
```

Два клиента, которые будут делать вызов к этому методу, не должны получать один и тот же ID, так как это приведет к тому, что данные соответствующих сущностей будут перезаписываться. Метод `nextEntity()` должен возвращать целочисленное значение и одновременно присваивать ему метку «используется». Однако при этом нарушится принцип разделения команд и запросов: метод будет возвращать информацию и выполнять задачи, тем самым явно влияя на состояние системы. Повторный вызов метода даст другой результат.

Можно постараться следовать принципу разделения, но, скорее всего, это слишком усложнит код[1]. В этом случае в целях рациональной реализации метода лучше этот принцип не использовать.

[1] Подробнее об этой ситуации и методах решения можно прочесть в статье Марка Симана (Mark Seeman. «CQS versus server generated Ids» (2014)), http://mng.bz/Q0nQ.

ЗАКЛЮЧЕНИЕ

- Метод-запрос — это метод, который можно использовать для извлечения некоторой информации. Методы-запросы должны иметь единый возвращаемый тип. Можно возвращать тип `null`, но лучше поискать альтернативы, например пустой объект или пустую коллекцию. Как вариант — выдавать исключение. Методы-запросы должны раскрывать минимально необходимые детали реализации объектов.
- Для вопросов, на которые требуется ответ, используйте специфичные методы и их возвращаемые типы. Задайте для этих методов абстракцию (интерфейс, в котором нет деталей реализации), если ответ на вопрос можно получить только при условии выхода приложения за границы системы.
- При тестировании сервисов, которые используют запросы для получения информации, замените их на заглушки или фиктивные объекты и не тестируйте реальные вызовы.

ОТВЕТЫ К УПРАЖНЕНИЯМ

1 Правильные ответы: **а**, **в** и **г**. В ответе б есть возвращаемый тип `void`, поэтому это не метод-запрос.

2 Правильные ответы: **б** и **в**. Метод-запрос не должен явно создавать побочный эффект. Метод-запрос всегда должен иметь определенный возвращаемый тип, хотя бы `null`, поэтому тип `void` недопустим.

3 Вариант ответа:

```
final class Line
{
    // ...

    public function amount(): Money
    {
        return new Money(
            line.quantity() * line.tariff()
        );
    }
}

final class Order
{
    // ...

    public function totalAmount(): Money
    {
```

Здесь можно удалить методы tariff() и quantity(), что позволит хранить данные внутри объекта

Можно также удалить метод lines(), оставив массив lines и объекты Line приватными

```
        totalAmount = new Money(0);

        foreach (this.lines() as line) {
            totalAmount = totalAmount.add(
                line.amount()
            );
        }

        return totalAmount;
    }
}
```

4 Правильные ответы: **б** и **в**. Абстрактный класс здесь не лучший вариант, так как он оставит часть реализации неопределенной, то же самое будет наблюдаться в любом его подклассе. Также нет необходимости предоставлять более одной реализации интерфейса.

7 Выполнение задач

В этой главе:

- ✓ Использование командных методов для выполнения задач
- ✓ Использование событий и прослушивателей событий для разделения больших задач
- ✓ Обработка ошибок в командных методах
- ✓ Добавление абстракций в команды
- ✓ Создание тестовых дублеров для вызовов командных методов

Помимо извлечения информации из объектов, их можно использовать для выполнения различных задач:

- отправки электронных уведомлений;
- сохранения записей в базе данных;
- смены пароля учетной записи пользователя;
- сохранения данных на диск;
- и т. д.

Далее мы рассмотрим правила для методов, выполняющих подобные задачи.

7.1. ИСПОЛЬЗУЙТЕ КОМАНДНЫЕ МЕТОДЫ С ИМЕНЕМ В ИМПЕРАТИВНОЙ ФОРМЕ

Мы уже говорили о том, что такое методы-запросы и как их использовать для извлечения информации. Методы-запросы имеют определенный возвращаемый тип и не вызывают побочных эффектов, поэтому их можно безопасно вызывать несколько раз, и после вызова состояние приложения меняться не будет.

Для выполнения задач предпочтительно использовать командный метод с возвращаемым типом `void`. Имя такого метода должно указывать на то, что клиент может приказать объекту выполнить соответствующую задачу. Подходящим именем будет вариант в императивной форме. В следующем листинге показаны несколько примеров.

Листинг 7.1. Командные методы с императивными именами

```
public function sendReminderEmail(
    EmailAddress recipient,
    // ...
): void {
    // ...
}

public function saveRecord(Record record): void
{
    // ...
}
```

7.2. ОГРАНИЧИВАЙТЕ ОБЛАСТЬ ВОЗДЕЙСТВИЯ КОМАНДНОГО МЕТОДА И ИСПОЛЬЗУЙТЕ СОБЫТИЯ ДЛЯ ВЫПОЛНЕНИЯ ВТОРОСТЕПЕННЫХ ЗАДАЧ

При выполнении задачи убедитесь, что метод берет на себя не слишком много функций. Ответы на вопросы ниже помогут определить, не слишком ли велик метод:

- Есть ли необходимость использовать «and» в имени метода, чтобы описать его основную задачу?
- Все ли строки кода решают основную задачу?
- Может ли часть работы осуществляться в фоновом режиме?

В листинге ниже показан метод с избыточным функционалом. Он не только меняет пароль пользователя, но и отправляет ему электронное письмо.

Листинг 7.2. Метод changePassword() выполняет слишком много задач

```
public function changeUserPassword(
    UserId userId,
    string plainTextPassword
): void {
    user = this.repository.getById(userId);
    hashedPassword = /* ... */;
    user.changePassword(hashedPassword);
    this.repository.save(user);
    this.mailer.sendPasswordChangedEmail(userId);
}
```

В этом сценарии ответы на все наводящие вопросы — «да»:

- Имя метода скрывает, что, помимо изменения пользовательского пароля, он еще и отправляет письмо. Правильнее было бы назвать этот метод `changeUserNameAndSendEmailAboutIt()`.
- Отправку письма нельзя считать основной задачей метода, его основная задача — изменение пароля.
- Письмо легко отправить при помощи другого процесса, который будет работать в фоновом режиме.

Чтобы исправить ситуацию, можно переместить код для отправки письма в новый метод `sendPasswordChangeEmail()`. Это переложит ответственность за вызов метода на клиента метода `changeUserPassword()`. По большому счету эти две задачи нельзя рассматривать по отдельности, но все равно не стоит смешивать их в одном методе.

Мы рекомендуем другое решение: использовать события в качестве связи между задачами изменения пароля и уведомления об этом пользователя.

Листинг 7.3. При помощи события задача делится на части

```
final class UserPasswordChanged
{
    private UserId userId;

    public function __construct(UserId userId)
    {
        this.userId = userId;
    }

    public function userId(): UserId
    {
```

Изменение пароля пользователем можно представить в виде объекта-события UserPasswordChanged

```
        return this.userId;
    }
}

public function changeUserPassword(
    UserId userId,
    string plainTextPassword
): void {
    user = this.repository.getById(userId);
    hashedPassword = /* ... */;
    user.changePassword(hashedPassword);
    this.repository.save(user);

    this.eventDispatcher.dispatch(
        new UserPasswordChanged(userId)
    );
}
```

После смены пароля будет инициировано событие UserPasswordChanged, чтобы другие сервисы могли на него отреагировать

```
final class SendEmail
{
    // ...

    public function whenUserPasswordChanged(
        UserPasswordChanged event
    ): void {
        this.mailer.sendPasswordChangedEmail(event.userId());
    }
}
```

SendEmail — это прослушиватель событий для UserPasswordChanged. Когда появится объект события, этот прослушиватель отправит электронное сообщение

Для обслуживания таких сервисов прослушивателей, как `SendEmail`, требуется диспетчер событий. В большинстве фреймворков уже есть диспетчеры событий, которые можно использовать, или же написать собственный простой диспетчер, как показано ниже.

Листинг 7.4. Пример реализации EventDispatcher

```
final class EventDispatcher
{
    private array listeners;

    public function __construct(array listenersByType)
    {
        foreach (listenersByType as eventType => listeners) {
            Assertion.string(eventType);
            Assertion.allIsCallable(listeners);
        }

        this.listeners = listenersByType;
    }

    public function dispatch(object event): void
    {
```

```
        foreach (this.listenersFor(event.className) as listener) {
            listener(event);
        }
    }

    private function listenersFor(string event): array
    {
        if (isset(this.listeners[event])) {
            return this.listeners[event];
        }

        return [];
    }
}

listener = new SendEmail(/* ... */);
dispatcher = new EventDispatcher([
    UserPasswordChanged.className =>
         [listener, 'whenUserPasswordChanged']
]);

dispatcher.dispatch(new UserPasswordChanged(/* ... */));
```

Так как SendEmail зарегистрирован как прослушиватель событий для UserPasswordChanged, отправка события такого типа спровоцирует вызов SendEmail.whenUserPasswordChanged()

Использование событий дает несколько преимуществ:

- Можно добавлять больше второстепенных функций, не меняя исходный метод.
- Исходный объект будет менее связанным, так как в него не внедряются зависимости, которые нужны только для второстепенных функций.
- Можно управлять поведением второстепенных функций в фоне.

Есть и недостаток использования событий: основная логика и второстепенные функции могут выполняться в разных частях кода, что усложняет понимание кода новыми членами команды. Чтобы этого избежать, сделайте следующее:

- Убедитесь, что все знают о том, что события уменьшают степень связанности компонентов приложения. Тот, кто будет пытаться понять, что делает данный код, будет ориентироваться по объектам-событиям и применять поисковую функциональность IDE, чтобы находить другие сервисы, использующие эти события.
- Убедитесь, что события всегда отправляются явно, как в листинге 7.3. Вызов к `EventDispatcher.dispatch()` — это сигнал, что сейчас произойдет событие.

УПРАЖНЕНИЕ

1 Какие части следующего командного метода можно рассматривать как второстепенные функции и переложить на прослушивателя событий?

```
final class RegisterUser
{
    // ...

    public function register(
        EmailAddress emailAddress
        PlainTextPassword plainTextPassword
    ): void {
        hashedPassword = this.passwordHasher
            .hash(plainTextPassword);

        userId = this.userRepository.nextIdentity();
        user = User.create(userId, emailAddress, hashedPassword);

        this.mailer.sendEmailAddressConfirmationEmail(
            emailAddress
        );

        this.userRepository.save(user);

        this.uploadService.preparePersonalUploadFolder(userId);
    }
}
```

а Хеширование незашифрованного пароля.

б Создание сущности `User`.

в Отправку уведомления о подтверждении адреса электронной почты.

г Сохранение сущности `User`.

д Подготовку персональной директории для пользователя.

7.3. СОЗДАВАЙТЕ СЕРВИС НЕИЗМЕНЯЕМЫМ ИЗНУТРИ И СНАРУЖИ

Мы уже рассмотрели правило запрета изменения зависимостей или конфигурации сервисов. После инстанцирования сервиса необходимо сделать так, чтобы его объект можно было многократно использовать для выполнения множества задач одинаковым образом, но с разными данными и в разном контексте. Риска изменения поведения сервиса при этом быть не должно.

Это правило справедливо для сервисов не только с методами-запросами, но и с командными методами.

Даже если вы не позволяете клиентам управлять зависимостями или конфигурацией сервиса, командный метод все равно может изменять состояние сервиса так, что его поведение при последующих вызовах будет другим. Например, сервис `Mailer` отправляет сообщение с подтверждением электронного адреса и при этом помнит пользователей, которые это сообщение уже получали. Неважно, сколько раз был вызван метод, — письмо будет отправлено всего один раз.

Листинг 7.5. Сервис Mailer, который хранит список получателей сообщения

```
final class Mailer
{
    private array sentTo = [];

    // ...

    public function sendConfirmationEmail(
        EmailAddress recipient
    ): void {
        if (in_array(recipient, this.sentTo)) {
            return;
        }

        // Отправить электронное письмо...

        this.sentTo[] = recipient;
    }
}

mailer = new Mailer(/* ... */);
recipient = EmailAddress.fromString('info@matthiasnoback.nl');

mailer.sendConfirmationEmail(recipient);

mailer.sendConfirmationEmail(recipient);
```

Электронное письмо не отправляется повторно

Будет отправлено письмо о подтверждении электронного адреса

Со вторым вызовом письмо повторно отправляться не будет

Убедитесь, что ни один из сервисов не изменяет внутреннее состояние, влияя на поведение сервиса.

Чтобы понять, насколько поведение сервиса соответствует ожиданиям, ответьте на вопрос, есть ли возможность повторно инстанцировать сервис при каждом вызове метода и будет ли при этом сохраняться прежнее поведение сервиса. В предыдущем классе `Mailer` очевидный ответ — «нет». Многократное инстанцирование сервиса приведет к тому, что одному получателю будет отправлено множество сообщений.

Если бы это был сервис `Mailer`, сохраняющий состояние, вопрос звучал бы так: как предотвратить дублирование вызовов в `sendConfirmationEmail()`? Представим, что клиент не может с этим справиться. И что, если вместо одного `EmailAddress` клиент будет предоставлять дедуплицированный список экземпляров `EmailAddess`? В таком случае можно использовать что-то вроде класса `Recipients`.

Листинг 7.6. Recipients предлагает список дедуплицированных email-адресов

```
final class Recipients
{
    /**
     * @var EmailAddress[]
     */
    private array emailEmailAddresses;

    /**
     * @return EmailAddress[]
     */
    public function uniqueEmailAddresses(): array
    {
        // Вернуть дедуплицированный список адресов...
    }
}

final class Mailer
{
    public function sendConfirmationEmails(
        Recipients recipients
    ): void {
        foreach (recipients.uniqueEmailAddresses()
            as emailAddress) {
            // Отправить электронное письмо...
        }
    }
}
```

Это определенно поможет решить проблему и снова преобразует сервис `Mailer` в сервис без состояния. Но вместо того чтобы позволять классу `Mailer` делать специальные вызовы к `uniqueEmailAddresses()`, мы будем использовать список `Recipients`, в котором не содержатся дубликаты электронных адресов. Этот инвариант предметной области лучше всего защитить внутри класса `Recipients`.

Листинг 7.7. Более эффективная реализация класса Recipients

```
final class Recipients
{
    /**
     * @var EmailAddress[]
```

```
     */
    private array emailAddresses;

    private function __construct(array emailAddresses)
    {
        this.emailAddresses = emailAddresses;
    }

    public static function emptyList(): Recipients
    {
        return new Recipients([]);
    }

    public function with(EmailAddress emailAddress): Recipients
    {
        if (in_array(emailAddress, this.emailAddresses)) {
            return this;
        }

        return new Recipients(
            array_merge(this.emailAddresses),
            [emailAddress]
        );
    }

    public function emailAddresses(): array
    {
        return this.emailAddresses;
    }
}
```

Всегда начинайте с пустого списка

Каждый раз, когда клиент захочет добавить электронный адрес, добавляться будет только адрес, которого еще нет в списке

Нет необходимости снова добавлять электронный адрес

Метод uniqueEmailAddress() больше не нужен

НЕИЗМЕНЯЕМЫЕ СЕРВИСЫ И КОНТЕЙНЕРЫ СЕРВИСОВ

Контейнеры сервисов часто проектируются таким образом, чтобы предоставить доступ ко всем экземплярам сервисов, как только они будут созданы. Это экономит время на инстанцирование одного и того же сервиса, если, например, он будет повторно использоваться как зависимость другого сервиса. Тем не менее, если сервис неизменяемый (как и должно быть), это распределение доступа и не понадобится. Можно просто заново инстанцировать сервис.

Конечно, в контейнере существуют сервисы, которые не нужно инстанцировать каждый раз, когда они используются как зависимость. Например, объект подключения к базе данных или ссылка на ресурс, которая создается один раз и становится доступной другим зависимым сервисам. В общем же случае сервисами делиться не стоит. Если вы до сих пор следовали всем рекомендациям, у вас должно все получиться, поскольку неизменяемые сервисы не обязательно должны использоваться совместно, хотя и могут это делать.

УПРАЖНЕНИЕ

2 Что помешает сервису быть неизменяемым?

- **а** Возможность внедрения зависимости через вызов метода.
- **б** Возможность изменения значений конфигурации через вызов метода.
- **в** Использование метода-запроса, который сам вызывает командный метод.
- **г** Слишком большое количество аргументов конструктора.
- **д** Изменение внутреннего состояния, когда клиент вызывает у сервиса метод.

7.4. КОГДА ЧТО-ТО ИДЕТ НЕ ТАК, ВЫДАВАЙТЕ ИСКЛЮЧЕНИЕ

Правило получения информации применимо и к методам, выполняющим задание. Когда что-то идет не так, не возвращайте специальное значение, чтобы обозначить это, а выдавайте исключение. Как мы уже обсуждали, у метода могут быть проверки предусловий, которые могут выдавать исключения `InvalidArgumentException` и `LogicException`. В других же сценариях отказа их появление предсказать заранее нельзя, поэтому будем выдавать `RuntimeException`. Мы уже обсуждали другие важные правила использования исключений в разделе 5.2.

УПРАЖНЕНИЯ

3 Какой тип исключения выдаст метод `save()`, если у него не получилось сохранить сущность `Product`, так как сущность с таким ID уже существует?

```
interface ProductRepository
{
    public function save(Product product): void;
}
```

- **а** `InvalidArgumentException`, так как клиент предоставил неверный аргумент `Product`.
- **б** `RuntimeException`, так как наличие или отсутствие сущности `Product` с данным ID невозможно установить только из аргументов.

4 Какой тип исключения выдаст `set()`, если для аргумента `key` будет передаваться пустая строка?

```
interface Cache
{
    public function set(string key, string value): void;
}
```

а `InvalidArgumentException`, так как клиент предоставил неверный аргумент.

б `RuntimeException`, так как во время исполнения клиент может сам решить, какое значение будет у аргумента `key`.

7.5. ИСПОЛЬЗУЙТЕ ЗАПРОСЫ ДЛЯ СБОРА ИНФОРМАЦИИ, А КОМАНДНЫЕ МЕТОДЫ — ДЛЯ ПОСЛЕДУЮЩИХ ДЕЙСТВИЙ

Когда мы обсуждали методы-запросы, то видели, что цепочка вызовов методов, которая начинается с метода-запроса, не должна содержать ни одного командного метода. Командный метод может вызывать побочные эффекты, и это нарушает правило о том, что у метода-запроса не должно быть побочных эффектов.

Теперь же мы рассматриваем командные методы, и стоит заметить, что для них подобного правила не существует. Если цепочка вызовов начинается с командного метода, вполне возможно встретить в ней методы-запросы. Например, метод `changeUserPassword()`, который мы рассматривали выше, начинается с запроса к репозиторию пользователей.

Листинг 7.8. changeUserPassword() начинается с запроса, а затем выполняет задачу

```
public function changeUserPassword(
    UserId userId,
    string plainTextPassword
): void {
    user = this.repository.getById(userId);
    hashedPassword = /* ... */;
    user.changePassword(hashedPassword);
    this.repository.save(user);
    this.eventDispatcher.dispatch(
        new UserPasswordChanged(userId)
    );
}
```

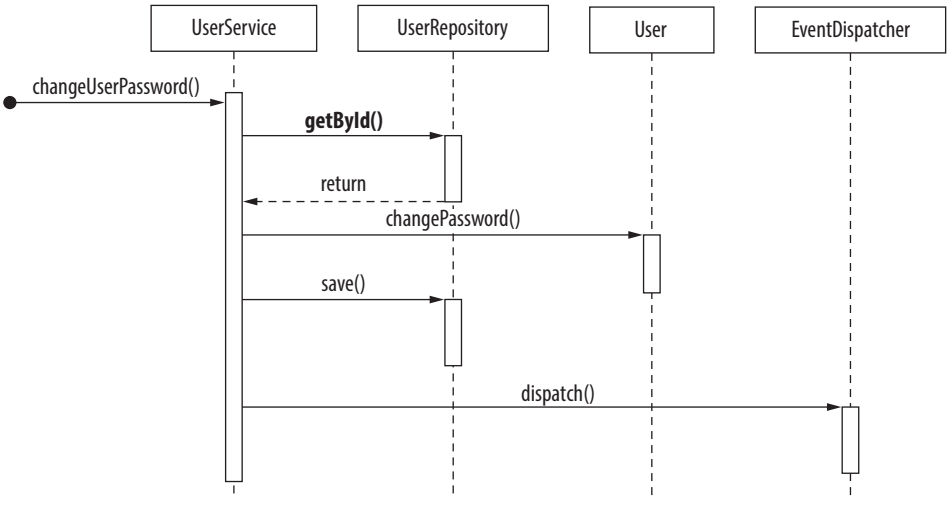


Рис. 7.1. В цепочке командного метода могут быть и запросы для получения дополнительной информации

Следующий метод в цепочке — это `changePassword()` у объекта пользователя, а за ним следует команда к репозиторию. Внутри репозитория могут также быть вызовы командных методов и, возможно, методов-запросов (рис. 7.1).

Рассматривая, как объекты вызывают друг у друга команды и запросы, имейте в виду паттерн, изображенный на рис. 7.2. Подобный паттерн вызовов часто указывает на небольшое взаимодействие объектов, которое могло произойти только в рамках вызываемого объекта. Рассмотрим следующий код:

```
if (obstacle.isOnTheRight()) {
    player.moveLeft();
} elseif (obstacle.isOnTheLeft()) {
    player.moveRight();
}
```

Ниже представлено улучшение этого фрагмента кода, в котором информация о том, какое действие следует предпринять, теперь находится полностью внутри одного объекта.

```
player.evade(obstacle);
```

Этот объект способен хранить такое знание внутри себя, и его реализацию можно свободно улучшать, если нужно добиться более сложного поведения.

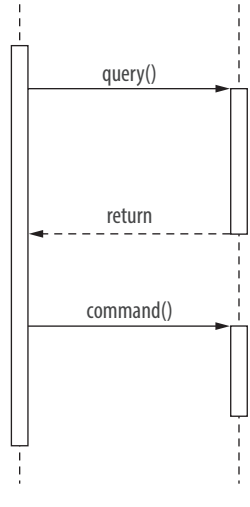


Рис. 7.2. Вызов метода-запроса, а затем командного метода в том же объекте

7.6. ЗАДАВАЙТЕ АБСТРАКЦИИ ДЛЯ КОМАНД, КОТОРЫЕ ВЫХОДЯТ ЗА ГРАНИЦЫ СИСТЕМЫ

Если в командном методе имеется код, который взаимодействует с частями приложения, находящимися за границами самого приложения (например, используются удаленный сервис, файловая система, системное устройство и т. п.), нужно создать абстракцию. В листинге ниже показан код, который отправляет сообщение в очередь, чтобы фоновые потребители настроились на важные события из основного приложения.

Листинг 7.9. SendMessageToRabbitMQ отправляет сообщения в очередь

```
final class SendMessageToRabbitMQ
{
    // ...

    public function whenUserChangedPassword(
        UserPasswordChanged event
    ): void {
        this.rabbitMqConnection.publish(
            'user_events',
            'user_password_changed',
            json_encode([
                'user_id' => (string)event.userId()
            ])
        );
    }
}
```

Метод `publish()` будет соединяться с сервером RabbitMQ и отправлять сообщение в его очередь, которая находится за пределами самого приложения, поэтому здесь нужна абстракция. Как мы уже говорили, это потребует наличия интерфейса и высокоуровневой концепции. Например, сохраняя представление о том, что мы хотим поставить сообщение в очередь, можно внедрить абстракцию `Queue`.

Листинг 7.10. Queue — абстракция, используемая SendMessageToRabbitMQ

```
interface Queue                                   <-- Queue — абстракция
{
    public function publishUserPasswordChangedEvent(
        UserPasswordChanged event
    ): void;
}

final class RabbitMQQueue implements Queue        <-- Стандартная реализация Queue — RabbitMQQueue, содержащая код, который мы уже видели
{
    // ...
```

```
    public function publishUserPasswordChangedEvent(
        UserPasswordChanged event
    ): void {
        this.rabbitMqConnection.publish(
            'user_events',
            'user_password_changed',
            json_encode([
                'user_id' => (string)event.userId()
            ])
        );
    }
}

final class SendMessageToRabbitMQ
{
    private Queue queue;

    public function __construct(Queue queue)
    {
        this.queue = queue;
    }

    public function whenUserPasswordChanged(
        UserPasswordChanged event
    ): void {
        this.queue.publishUserPasswordChangedEvent(event);
    }
}
```

Прослушиватель событий, который должен отправлять сообщение в очередь; когда появится событие UserPasswordChanged, будет использовать новую абстракцию в качестве зависимости

Первым шагом было создание *абстракции*. Когда вы начнете добавлять больше методов по типу `publish…Event()` в `Queue`, вы заметите нечто общее в этих методах. Затем можно применить *обобщение*, чтобы сделать методы более универсальными. Возможно, понадобится реализовать стандартный интерфейс для всех событий.

Листинг 7.11. Интерфейс CanBePublished для отправляемых событий

```
interface CanBePublished
{
    public function queueName(): string;
    public function eventName(): string;
    public function eventData(): array;
}

final class RabbitMQQueue implements Queue
{
    // ...

    public function publish(CanBePublished event): void
    {
        this.rabbitMqConnection.publish(
            event.queueName(),
```

```
            event.eventName(),
            json_encode(event.eventData())
        );
    }
}
```

В целом рекомендуется начинать с абстракции и не переходить к обобщению, пока вы не увидите три ситуации, в которых обобщение интерфейсов и типов объектов приведет к упрощению. Это поможет избежать слишком раннего создания абстракции и необходимости пересматривать интерфейс и его реализации каждый раз, когда требуется поддержка этой абстракцией.

УПРАЖНЕНИЕ

5 Почему задача сохранения сущности в базу данных требует абстракции?

- **а** Потому что эта сущность может исчезнуть.
- **б** Потому что наличие абстракции позволяет менять реализацию в тестовом сценарии.
- **в** Потому что абстракция может понадобиться для хранения других типов данных.
- **г** Потому что абстракция использует высокоуровневую концепцию, которая объясняет, что она выполняет, и облегчает понимание кода, так как в этом случае можно игнорировать низкоуровневые детали реализации.

7.7. ПРОВЕРЯЙТЕ ИМИТАЦИЕЙ (MOCK) ТОЛЬКО КОМАНДНЫЕ МЕТОДЫ

Мы уже говорили, что методы-запросы нельзя проверять при помощи имитаций (mock). В модульном тесте не требуется проверять количество совершенных вызовов метода. Запросы должны работать без побочных эффектов, чтобы при необходимости их можно было выполнять неограниченное количество раз. Если реализация будет способна на такое, это улучшит стабильность тестов. Если вы решите вызвать метод дважды вместо того, чтобы фиксировать результат тестирования в переменной, тест не завершится ошибкой.

Тем не менее, когда командный метод вызывает второй командный метод, может потребоваться имитировать последний. В конце концов, эта команда должна выполняться хотя бы раз (потому что ее надо проверить и она является частью цепочки выполняемых команд), но и больше одного раза выполнять ее не нужно (так как вам не нужно, чтобы побочные эффекты возникали повторно). Это показано в листинге ниже.

Листинг 7.12. Модульное тестирование ChangePasswordService при помощи имитации

```
final class ChangePasswordService
{
    private EventDispatcher eventDispatcher;
    // ...

    public function __construct(
        EventDispatcher eventDispatcher,
        // ...
    ) {
        this.eventDispatcher = eventDispatcher;
        // ...
    }

    public function changeUserPassword(
        UserId userId,
        string plainTextPassword
    ): void {
        // ...

        this.eventDispatcher.dispatch(
            new UserPasswordChanged(userId)
        );
    }
}

/**
 * @test
 */
public function it_dispatches_a_user_password_changed_event(): void
{
    userId = /* ... */;

    eventDispatcherMock = this.createMock(EventDispatcher.className);
    eventDispatcherMock
        .expects(this.once())
        .method('dispatch')
        .with(new UserPasswordChanged(userId));

    service = new ChangePasswordService(eventDispatcherMock, /* ... */);

    service.changeUserPassword(userId, /* ... */);
}
```

Здесь задается настоящий mock-объект: проверяется количество ожидаемых вызовов метода (один) и то, с какими аргументами метод будет вызван. Возвращаемое значение не проверяется, так как dispatch() — это командный метод

В конце этого теста нет обычных проверок, так как mock-объект сам по себе будет проверять на соответствие поведения метода ожиданиям. Тестовый фреймворк будет просить все mock-объекты, созданные для тест-кейса, выполнить эти проверки.

Если вы предпочитаете иметь в тесте явные утверждения, используйте *объект-шпион* (spy) в качестве тестового дублера для `EventDispatcher`. В самой

обобщенной форме он будет запоминать все сделанные к нему вызовы методов, а также аргументы, которые при этом использовались. Но нас устроит и простая реализация `EventDispatcher`.

Листинг 7.13. Шпион для EventDispatcher

```
final class EventDispatcherSpy implements EventDispatcher
{
    private array events = [];

    public function dispatch(object event): void
    {
        this.events[] = event;
    }

    public function dispatchedEvents(): array
    {
        return this.events;
    }
}

/**
 * @test
 */
public function it_dispatches_a_user_password_changed_event(): void
{
    // ...
    eventDispatcher = new EventDispatcherSpy();
    service = new ChangePasswordService(eventDispatcher, /* ... */);

    service.changeUserPassword(userId, /* ... */);

    assertEquals(
        [
            new UserPasswordChanged(userId)
        ],
        eventDispatcher.dispatchedEvents()
    );
}
```

Шпион отслеживает список событий, которые к нему отправляются

Теперь можно объявить проверку утверждений вместо того, чтобы ждать, пока тестовый фреймворк проверит вызов метода на объекте-имитации

УПРАЖНЕНИЯ

6 Рассмотрим следующий интерфейс:

```
interface UserRepository
{
    public function save(User user): void;
}
```

Какой тип тестового дублера использовать в тесте для класса, который вызывает метод `save()` на зависимости `UserRepository`?

а dummy (пустышка).

б stub (заглушка).

в fake (фиктивный объект).

г mock (имитация).

д spy (шпион).

7 Рассмотрим следующий интерфейс:

```
interface UserRepository
{
    public function getById(UserId userId): User;
}
```

Какой тип тестового дублера использовать в тесте для класса, который вызывает метод `getById()` на зависимости `UserRepository`?

а dummy (пустышка).

б stub (заглушка).

в fake (фиктивный объект).

г mock (имитация).

д spy (шпион).

ЗАКЛЮЧЕНИЕ

- Командные методы должны использоваться для выполнения задач. Этим методам необходимо присваивать названия в форме императива («сделай это», «сделай то»), и область их воздействия должна быть ограничена. Выделяйте основную задачу и второстепенные. Передавайте события, чтобы другие сервисы могли выполнять дополнительные задачи. Пока эти задачи выполняются, командный метод может совершать вызовы методов-запросов, чтобы получать любую необходимую информацию.
- Сервис должен быть неизменяемым снаружи и изнутри. Как и с сервисами для извлечения данных, необходимо обеспечить возможность многократного использования сервисов для выполнения задач. Если что-то пойдет не так при выполнении задачи, выдавайте исключение (как только о проблеме станет известно).

- Задавайте абстракцию для команд, которые выходят за границы приложения (команд, которые обращаются к удаленному сервису, базе данных и т. п.). При тестировании командных методов, которые сами также вызывают командные методы, используйте имитации объектов (mock) или шпионы (spy) для проверки вызовов к этим методам.

ОТВЕТЫ К УПРАЖНЕНИЯМ

1 Правильный ответ: **в** и **д**. Все остальные варианты нужно рассматривать как часть основного действия. Действия **в** и **д** являются второстепенными действиями, или эффектами, по отношению к основному.

2 Правильный ответ: **а**, **б** и **д**. Без зависимостей или значений конфигурации сервис станет изменяемым. Количество аргументов конструкторов не влияет на неизменяемость. То же самое верно и при взаимодействии с другими объектами.

3 Правильный ответ: **б**. Объяснение содержится в самом ответе.

4 Правильный ответ: **а**. Объяснение содержится в самом ответе.

5 Правильный ответ: **б** и **г**. Объяснение содержится в самом ответе. Ответ а неверный, так как если избавиться от сущности, то вы избавитесь и от ее репозитория. Ответ в неверный, так как это потребует создания обобщений в репозитории, что излишне.

6 Правильный ответ: **г** и **д**. `save()` — это командный метод, поэтому мы используем имитацию объекта mock (чтобы проверить, что метод был вызван) или объект-шпион spy (чтобы узнать, был ли вызван метод).

7 Правильный ответ: **а**, **б** и **в**. `getByIt()` — метод-запрос, поэтому мы передаем пустышку (dummy, нефункциональный объект корректного типа), заглушку (stub, объект корректного типа, который может возвращать заранее сконфигурированное значение) или фиктивный объект (fake, более сложный объект с собственной логикой). Мы не хотим, чтобы проверки совершались на вызовах к настоящим ресурсам, поэтому не используем имитацию (mock) или шпион (spy).

8 Разделение функций

В этой главе:

- ✓ Различия между моделями чтения и записи
- ✓ Задание отдельных репозиториев для моделей чтения и записи
- ✓ Разработка моделей чтения для конкретных задач
- ✓ Построение модели чтения из событий или общего источника данных

Мы уже рассматривали, как можно использовать объекты для получения информации или выполнения задач. Методы извлечения информации называются методами-запросами, а методы, выполняющие задачи, — командными.

Объекты-сервисы способны сочетать в себе обе эти функции. Например, репозиторий (подобный тому, что приведен в следующем листинге) способен выполнять задачу как сохранения сущности в базе данных, так и ее извлечения.

Листинг 8.1. PurchaseOrderRepository может сохранять и извлекать PurchaseOrder

```
interface PurchaseOrderRepository
{
    /**
     * @throws CouldNotSavePurchaseOrder
```

```
     */
    public function save(PurchaseOrder purchaseOrder): void;

    /**
     * @throws CouldNotFindPurchaseOrder
     */
    public function getById(int purchaseOrderId): PurchaseOrder;
}
```

Операции сохранения и извлечения сущности в определенной степени обратны по отношению друг к другу, поэтому вполне естественно позволить одному объекту выполнять их обе. Однако в других случаях выполнение задач и извлечение информации лучше разделять между разными объектами.

8.1. ОТДЕЛЯЙТЕ МОДЕЛИ ЗАПИСИ ОТ МОДЕЛЕЙ ЧТЕНИЯ

Как мы уже знаем, существуют сервисы и прочие объекты. Некоторые из этих прочих объектов можно охарактеризовать как сущности, моделирующие определенное понятие предметной области. При этом они содержат релевантные данные и предлагают способы эффективного управления этими данными. Сущности также способны предоставлять данные, позволяя клиентам извлекать из них информацию, будь то открытые внутренние данные (например, дата размещения заказа) или вычисляемые (например, общая сумма заказа).

На практике различные клиенты используют сущности по-разному. Некоторым требуется манипулировать данными объекта, используя его командные методы, в то время как другим просто нужно получить часть информации с помощью методов-запросов. Но все клиенты будут совместно использовать один и тот же объект и потенциально будут иметь доступ ко всем методам, даже к ненужным или к тем, к которым им не следует иметь доступа.

Никогда не передавайте потенциально изменяемый объект клиенту, которому не разрешено его менять. Даже если клиент не изменит его прямо сейчас, однажды он все же может это сделать, и тогда будет трудно выяснить, что произошло. Вот почему первое, что нужно сделать в рамках улучшения дизайна объекта, — это отделить *модель записи* от *модели чтения*.

Рассмотрим, как это сделать, на примере объекта `PurchaseOrder`. Заказ на покупку представляет собой факт приобретения компанией продукта у одного из поставщиков. Как только товар получен, он отправляется на склад компании и хранится там. Мы будем использовать этот пример до конца главы и подумаем, как его можно улучшить.

Листинг 8.2. Сущность PurchaseOrder

```
final class PurchaseOrder
{
    private int purchaseOrderId;
    private int productId;
    private int orderedQuantity;
    private bool wasReceived;

    private function __construct()
    {
    }

    public static function place(
        int purchaseOrderId,
        int productId,
        int orderedQuantity
    ): PurchaseOrder {
        purchaseOrder = new PurchaseOrder();

        purchaseOrder.productId = productId;
        purchaseOrder.orderedQuantity = orderedQuantity;
        purchaseOrder.wasReceived = false;

        return purchaseOrder;
    }

    public function markAsReceived(): void
    {
        this.wasReceived = true;
    }

    public function purchaseOrderId(): int
    {
        return this.purchaseOrderId;
    }

    public function productId(): int
    {
        return this.productId;
    }

    public function orderedQuantity(): int
    {
        return this.orderedQuantity;
    }

    public function wasReceived(): bool
    {
        return this.wasReceived;
    }
}
```

Для краткости использованы значения примитивных типов; на практике же рекомендуется использовать объекты-значения

В текущей реализации объект `PurchaseOrder` предоставляет методы для создания объекта и управления им (`place()` и `markasreceived()`), а также для извлечения из него информации (`ProductID()`, `orderedQuantity()` и `wasreceived()`).

Теперь взгляните, как разные клиенты используют эту сущность. Прежде всего сервис `ReceiveItems` будет вызван из контроллера с передачей необработанного идентификатора заказа.

Листинг 8.3. Сервис ReceiveItems

```
final class ReceiveItems
{

    private PurchaseOrderRepository repository;

    public function __construct(PurchaseOrderRepository repository)
    {
        this.repository = repository;
    }

    public function receiveItems(int purchaseOrderId): void
    {
        purchaseOrder = this.repository.getById(purchaseOrderId);

        purchaseOrder.markAsReceived();

        this.repository.save(purchaseOrder);
    }
}
```

Обратите внимание, что сервис не использует ни один из геттеров в `PurchaseOrder`. Речь идет только об изменении состояния сущности.

Далее посмотрим на контроллер, который отображает структуру данных в кодировке JSON с подробным описанием, сколько продуктов у компании на складе.

Листинг 8.4. Класс StockReportController

```
final class StockReportController
{
    private PurchaseOrderRepository repository;

    public function __construct(PurchaseOrderRepository repository)
    {
        this.repository = repository;
    }

    public function execute(Request request): Response
    {
        allPurchaseOrders = this.repository.findAll();
```

```
        stockReport = [];

        foreach (allPurchaseOrders as purchaseOrder) {
            if (!purchaseOrder.wasReceived()) {
                continue;
            }

            if (!isset(stockReport[purchaseOrder.productId()] )) {
                stockReport[purchaseOrder.productId()] = 0;
            }

            stockReport[purchaseOrder.productId()]
                += purchaseOrder.orderedQuantity;
        }

        return new JsonResponse(stockReport);
    }
}
```

Мы пока не получили товары, так что еще рано прибавлять их к количеству на складе

Ранее нам этот товар не встречался...

Добавить заказанное (и полученное) количество к количеству на складе

Контроллер не вносит изменений в `PurchaseOrder`. От каждого заказа ему просто требуется фрагмент информации. Другими словами, его не интересует часть сущности, относящаяся к записи. Ему нужна только часть для чтения. Помимо того что нежелательно предоставлять клиенту больше информации о поведении, чем необходимо, не очень эффективно перебирать заказы на покупку за все время, чтобы узнать, сколько товара имеется на складе.

Решение — разделить функции внутри сущности. Сначала создадим новый объект, который можно использовать для получения информации о заказе. Назовем его `PurchaseOrderForStockReport`.

Листинг 8.5. Класс PurchaseOrderForStockReport

```
final class PurchaseOrderForStockReport
{
    private int productId;
    private int orderedQuantity;
    private bool wasReceived;

    public function __construct(
        int productId,
        int orderedQuantity,
        bool wasReceived
    ) {
        this.productId = productId;
        this.orderedQuantity = orderedQuantity;
        this.wasReceived = wasReceived;
    }

    public function productId(): ProductId
    {
        return this.productId;
```

```
    }

    public function orderedQuantity(): int
    {
        return this.orderedQuantity;
    }

    public function wasReceived(): bool
    {
        return this.wasReceived;
    }
}
```

Этот новый объект `PurchaseOrderForStockReport` можно использовать внутри контроллера, как только появится репозиторий, который может его предоставить. Поспешным и не очень аккуратным решением было бы позволить `PurchaseOrder` возвращать экземпляр `PurchaseOrderForStockReport` на основе своих внутренних данных.

Листинг 8.6. Быстрое решение: PurchaseOrder генерирует отчет

```
final class PurchaseOrder
{
    private int purchaseOrderId
    private int productId;
    private int orderedQuantity;
    private bool wasReceived;

    // ...

    public function forStockReport(): PurchaseOrderForStockReport
    {
        return new PurchaseOrderForStockReport(
            this.productId,
            this.orderedQuantity,
            this.wasReceived
        );
    }
}

final class StockReportController
{
    private PurchaseOrderRepository repository;

    public function __construct(PurchaseOrderRepository repository)
    {
        this.repository = repository;
    }

    public function execute(Request request): Response
    {
        allPurchaseOrders = this.repository.findAll();
```

Мы все еще загружаем сущности PurchaseOrder

```
        forStockReport = array_map(
            function (PurchaseOrder purchaseOrder) {
                return purchaseOrder.forStockReport();
            },
            allPurchaseOrders
        );

    // ...

    }
}
```

Мы немедленно преобразуем их в экземпляры PurchaseOrderForStockReport

Теперь можно удалить практически все методы-запросы (`ProductID()`, `orderedQuantity()` и `wasreceived()`) из исходной сущности `PurchaseOrder`. Это делает ее правильной моделью записи; она больше не используется клиентами, которые просто хотят получить от нее информацию.

Листинг 8.7. PurchaseOrder с удаленными геттерами

```
final class PurchaseOrder
{
    private int purchaseOrderId
    private int productId;
    private int orderedQuantity;
    private bool wasReceived;

    private function __construct()
    {
    }

    public static function place(
        int purchaseOrderId,
        int productId,
        int orderedQuantity
    ): PurchaseOrder {
        purchaseOrder = new PurchaseOrder();

        purchaseOrder.productId = productId;
        purchaseOrder.orderedQuantity = orderedQuantity;

        return purchaseOrder;
    }

    public function markAsReceived(): void
    {
        this.wasReceived = true;
    }
}
```

Как видно из следующего листинга, удаление этих методов-запросов не нанесет никакого вреда существующим клиентам `PurchaseOrder`, таким как рассмотренный

ранее сервис `ReceiveItems`, которые используют этот объект в качестве модели записи.

Листинг 8.8. Существующие клиенты используют PurchaseOrder в качестве модели записи

```
final class ReceiveItems
{
    // ...

    public function receiveItems(int purchaseOrderId): void
    {
        purchaseOrder = this.repository.getById(
            PurchaseOrderId.fromInt(purchaseOrderId)
        );

        purchaseOrder.markAsReceived();

        this.repository.save(purchaseOrder);
    }
}
```

Этот сервис не использует методы-запросы PurchaseOrder

МЕТОДЫ-ЗАПРОСЫ НЕ ЗАПРЕЩЕНЫ

Некоторые клиенты используют объект в качестве модели записи, но им все равно нужно извлекать из него определенную информацию. Она нужна, чтобы принимать на ее основе решения, выполнять дополнительные проверки и т. д. Не думайте, что в подобных случаях не следует добавлять методы-запросы. Речь об их запрете не идет. Цель этой главы — показать, что клиенты, которые используют сущность исключительно для извлечения информации, должны использовать отдельную модель чтения вместо модели записи.

УПРАЖНЕНИЯ

1 Является ли объект `SalesInvoice` в следующем коде моделью записи или же это модель чтения?

```
public function finalize(SalesInvoiceId salesInvoiceId): void
{
    salesInvoice = salesInvoiceRepository.getById(salesInvoiceId);

    if (salesInvoice.wasCancelled()) {
        throw new CanNotFinalizeSalesInvoice
            ::becauseItWasAlreadyCancelled(salesInvoiceId);
    }
```

```
        salesInvoice.finalize();

        eventDispatcher.dispatchAll(salesInvoice.recordedEvents());

        salesInvoiceRepository.save(salesInvoice);
    }
```

а Модель чтения.

б Модель записи.

2 Является ли объект `meetup` в следующем коде моделью записи или же это модель чтения?

```
public function meetupDetailsAction(Request request): Response
{
    meetup = meetupRepository.getById(request.get('meetupId'));

    return this.templateRenderer.render(
        'meetup-details.html.twig', [
            'meetup' => meetup
        ]
    );
}
```

а Модель чтения.

б Модель записи.

8.2. СОЗДАВАЙТЕ МОДЕЛИ ЧТЕНИЯ С УЧЕТОМ СЦЕНАРИЕВ ИХ ИСПОЛЬЗОВАНИЯ

В предыдущем разделе мы разделили сущность `PurchaseOrder` на модели записи и чтения. Мы оставили модели записи ее старое название, а модель чтения назвали `PurchaseOrderForStockReport`. Дополнительная квалификация `ForStockreport` указывает на то, что данный объект теперь служит определенной цели. Он пригоден для использования в очень специфическом контексте, а именно для упорядочивания данных с целью создания полезного отчета о товарных остатках. Предлагаемое решение пока неоптимально, поскольку контроллеру все еще необходимо загружать все объекты `PurchaseOrder` и преобразовывать их в экземпляры `PurchaseOrderForStockReport`, вызывая для них функцию `forStockReport()`, как показано в следующем листинге. Это значит, что клиент по-прежнему имеет доступ к модели записи, хотя изначально мы стремились этого избежать.

Листинг 8.9. Создание отчета о товарных остатках по-прежнему зависит от модели записи

```
public function execute(Request request): Response
{
    allPurchaseOrders = this.repository.findAll();

    forStockReport = array_map(
        function (PurchaseOrder purchaseOrder) {
            return purchaseOrder.forStockReport();
        },
        allPurchaseOrders
    );

    // ...
}
```

Мы по-прежнему полагаемся на экземпляры PurchaseOrder

Еще один аспект дизайна не совсем хорош: несмотря на то что теперь у нас есть объекты `PurchaseOrderForStockReport`, нам все равно нужно перебирать их и создавать новую структуру данных, прежде чем появится возможность представить данные пользователю. Но что, если бы у нас был объект, структура которого соответствовала бы сценарию его использования? Что касается имени этого объекта, то в названии модели чтения уже есть подсказка (`ForStockReport`). Итак, назовем этот новый объект `StockReport` и предположим, что он уже существует. Контроллер стал бы намного проще, как это показано в следующем листинге.

Листинг 8.10. StockReportController способен напрямую получать отчет о товарных остатках

```
final class StockReportController
{
    private StockReportRepository repository;

    public function __construct(StockReportRepository repository)
    {
        this.repository = repository;
    }

    public function execute(Request request): Response
    {
        stockReport = this.repository.getStockReport();

        return new JsonResponse(stockReport.asArray());
    }
}
```

Ожидается, что asArray() вернет массив, подобный тому, который мы ранее создавали вручную

Помимо `StockReport`, мы можем создавать любое количество моделей чтения, которые соответствуют каждому из конкретных вариантов использования приложения. Например, можно создать модель чтения, которая нужна для перечисления

заказов. В ней будут указаны только идентификатор и дата его создания. Затем можно создать отдельную модель чтения, которая предоставляет все подробности, необходимые для отображения формы, чтобы пользователь мог обновить информацию и т. д.

За кулисами `StockrepOrtrePosItory` все еще может создавать объект `Stockreport` на основе объектов `PurchaseOrderForStock`, предоставляемых сущностями модели записи. Но есть гораздо лучшие и более эффективные альтернативы. Мы рассмотрим некоторые из них в следующих разделах.

8.3. СОЗДАВАЙТЕ МОДЕЛИ ЧТЕНИЯ НЕПОСРЕДСТВЕННО ИЗ ИХ ИСТОЧНИКА ДАННЫХ

Вместо создания модели `StockReport` из объектов `PurchaseOrderForStock` можно перейти непосредственно к источнику данных — базе данных, в которой приложение хранит заказы на покупку. Если это реляционная база данных, в ней может быть таблица с именем `purchase_orders`, со столбцами `purchase_order_id`, `product_id`, `ordered _quantity` и `was_received`. Если это так, `StockReportRepository` не нужно загружать другой объект до создания объекта `Stockreport`. Можно выполнить единственный SQL-запрос и использовать его для создания `StockReport`.

Листинг 8.11. StockReportSqlRepository создает отчет о товарных остатках с использованием обычного SQL

```
final class StockReportSqlRepository implements StockReportRepository
{
    public function getStockReport(): StockReport
    {
        result = this.connection.execute(
            'SELECT ' .
            ' product_id, ' .
            ' SUM(ordered_quantity) as quantity_in_stock ' .
            'FROM purchase_orders ' .
            'WHERE was_received = 1 ' .
            'GROUP BY product_id'
        );

        data = result.fetchAll();

        return new StockReport(data);
    }
}
```

Создание моделей чтения непосредственно из источника данных модели записи обычно довольно эффективно с точки зрения производительности. Оно также

эффективно с точки зрения затрат на разработку и обслуживание. Решение становится менее эффективным, если модель записи часто меняется или если исходные данные трудно использовать в чистом виде без предварительной интерпретации.

8.4. ПОСТРОЕНИЕ МОДЕЛЕЙ ЧТЕНИЯ ИЗ СОБЫТИЙ ПРЕДМЕТНОЙ ОБЛАСТИ

Один из недостатков создания модели чтения `StockReport` непосредственно из данных модели записи — то, что приложение будет выполнять вычисления снова и снова каждый раз, когда пользователь будет запрашивать отчет об остатках. Хотя выполнение SQL-запроса и не будет занимать много времени (до тех пор пока таблица не станет слишком объемной), в ряде случаев потребуется использовать другой подход для создания моделей чтения.

Посмотрим еще раз на результат SQL-запроса из предыдущего примера (табл. 8.1).

Таблица 8.1. Результат SQL-запроса для создания отчета о товарных остатках

product_id	quantity_in_stock
123	10
124	5

Как еще можно получить числа во втором столбце, не просматривая все записи в таблице `purchase_orders` и не суммируя их значения `ordered_quantity`?

Что, если бы мы сидели с листом бумаги рядом с пользователем и всякий раз, когда он отмечает заказ как полученный, записывали идентификатор и количество полученных товаров? Итоговый список выглядел бы как таблица 8.2.

Таблица 8.2. Результат записи каждого полученного продукта

product_id	received
123	2
124	4
124	1
123	8

Теперь вместо того чтобы добавлять несколько строк для одного и того же продукта, можно отыскать строку с только что полученным продуктом и добавить полученное количество к числу, которое уже имеется в столбце `received`, как в табл. 8.3.

Таблица 8.3. Результат объединения полученного количества для каждого продукта

product_id	received
123	2 + 8
124	4 + 1

Вычисления приведут к тому же результату, что и при использовании запроса `SUM`.

Вместо того чтобы сидеть с бумагой рядом с пользователем, мы будем контролировать сущность `PurchaseOrder`, чтобы узнать, когда пользователь помечает ее как полученную. Это можно делать, записывая и отправляя *события предметной области*. Такой метод мы уже обсуждали в разделе 4.12.

Прежде всего нужно разрешить `PurchaseOrder` записывать событие, указывающее, что заказанные товары были получены.

Листинг 8.12. Сущности PurchaseOrder сохраняют события PurchaseOrderReceived

```
final class PurchaseOrderReceived  <-- Это новое событие предметной области
{
    private int purchaseOrderId;
    private int productId;
    private int receivedQuantity;

    public function __construct(
        int purchaseOrderId,
        int productId,
        int receivedQuantity
    ) {
        this.purchaseOrderId = purchaseOrderId;
        this.productId = productId;
        this.receivedQuantity = receivedQuantity;
    }

    public function productId(): int
    {
        return this.productId;
    }

    public function receivedQuantity(): int
    {
        return this.receivedQuantity;
    }
}

final class PurchaseOrder
{
    private array events = [];
```

```
    // ...

    public function markAsReceived(): void
    {
        this.wasReceived = true;

        this.events[] = new PurchaseOrderReceived(
            this.purchaseOrderId,
            this.productId,
            this.orderedQuantity
        );
    }

    public function recordedEvents(): array
    {
        return this.events;
    }
}
```

Событие предметной области записывается внутри PurchaseOrder

Вызов `markAsReceived()` теперь добавляет объект события `PurchaseOrderReceived` во внутренний список сохраненных событий. Эти события могут быть извлечены и переданы диспетчеру событий, как в следующем сервисе `ReceiveItems`.

Листинг 8.13. ReceiveItems отправляет любое записанное событие предметной области

```
final class ReceiveItems
{
    // ...

    public function receiveItems(int purchaseOrderId): void
    {
        // ...

        this.repository.save(purchaseOrder);

        this.eventDispatcher.dispatchAll(
            purchaseOrder.recordedEvents()
        );
    }
}
```

Прослушиватель событий, зарегистрированный для этого конкретного события, способен извлекать соответствующие данные из объекта события и обновлять собственный закрытый список продуктов и их количества на складе. Например, он мог бы создать отчет о товарных остатках, поддерживая собственную таблицу `stock_report` со строками для каждого продукта. Для этого он должен обрабатывать входящие события `PurchaseOrderReceived` и создавать новые строки либо обновлять уже существующие в `stock_report`.

Листинг 8.14. Использование события для обновления таблицы stock_report

```
final class UpdateStockReport
{
    public function whenPurchaseOrderReceived(
        PurchaseOrderReceived event
    ): void {
        this.connection.transactional(function () {
            try {
                this.connection                    ← Выясняем, существует ли строка для товара
                    .prepare(
                        'SELECT quantity_in_stock ' .
                        'FROM stock_report ' .
                        'WHERE product_id = :productId FOR UPDATE'
                    )
                    .bindValue('productId', event.productId())
                    .execute()
                    .fetch();
                                                   ← Если для этого товара уже существует строка,
                this.connection                      обновляем ее и увеличиваем количество на складе
                    .prepare(
                        'UPDATE stock_report ' .
                        'SET quantity_in_stock = ' .
                        ' quantity_in_stock + :quantityReceived ' .
                        'WHERE product_id = :productId'
                    )
                    .bindValue(
                        'productId',
                        event.productId()
                    )
                    .bindValue(
                        'quantityReceived',
                        event.quantityReceived()
                    )
                    .execute();
            } catch (NoResult exception) {         ← В противном случае создаем новую
                 this.connection                     строку и устанавливаем начальное
                    .prepare(                        значение для количества на складе
                        'INSERT INTO stock_report ' .
                        ' (product_id, quantity_in_stock) ' .
                        'VALUES (:productId, :quantityInStock)'
                    )
                    .bindValue(
                        'productId',
                        event.productId()
                    )
                    .bindValue(
                        'quantityInStock',
                        event.quantityReceived()
                    )
                    .execute();
            }
        });
    }
}
```

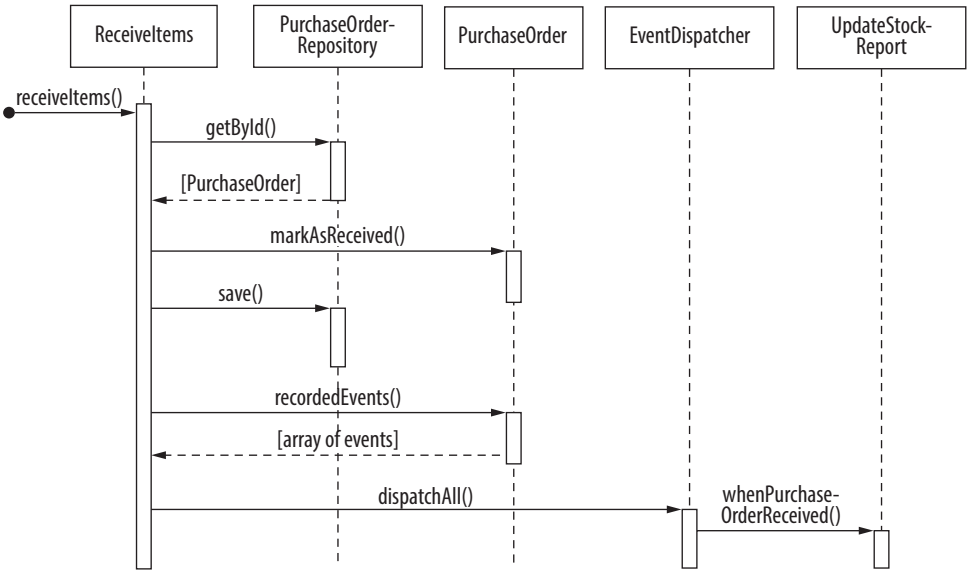


Рис. 8.1. Сервис Receiveitems вносит изменения в модель записи PurchaseOrder, а затем отправляет события в EventDispatcher, чтобы другие сервисы, такие как UpdateStockReport, могли отслеживать эти изменения

Как только у нас появится отдельный источник данных для отчета об остатках, мы сможем сделать `StockReportSqlRepository` еще проще, потому что вся информация уже имеется в таблице `stock_reports`.

Листинг 8.15. Запрос в StockReportSqlRepository теперь намного проще

```
final class StockReportSqlRepository implements StockReportRepository
{
    public function getStockReport(): StockReport
    {
        result = this.connection.execute(
            'SELECT * FROM stock_report'
        );

        data = result.fetchAll();

        return new StockReport(data);
    }
}
```

Такого рода упрощение повышает эффективность запросов в модели чтения. Однако с точки зрения затрат на разработку и обслуживание использование событий предметной области для создания моделей чтения обходится дороже.

Как видно из примеров в этом разделе, в данном случае задействовано больше механизмов. Если что-то изменится в событии, потребуется адаптировать другие части, которые от него зависят. Если один из прослушивателей событий выйдет из строя, придется исправлять ошибку и перезапускать его, что потребует дополнительных усилий с точки зрения инструментов и операций.

А ЧТО НАСЧЕТ ПОРОЖДЕНИЯ СОБЫТИЙ?

Все становится еще сложнее, если помимо использования событий для создания моделей чтения использовать их для реконструирования моделей записи. Этот метод называется *порождением событий*, и он очень хорошо согласуется с идеей отделения моделей записи от моделей чтения. Однако, как показано в этой главе, не следует применять порождение событий, если вы просто ищете лучший способ разделить функции между объектами. Можно предоставить клиентам, желающим только получить информацию из сущности, отдельную модель чтения, используя любой из описанных здесь методов.

УПРАЖНЕНИЕ

3 Составьте список событий предметной области, которые понадобятся для создания модели чтения для корзины покупок со следующими функциями:

- **а** Пользователь может добавить продукт.
- **б** Пользователь может удалить продукт.
- **в** Пользователь может изменить количество продукта.

ЗАКЛЮЧЕНИЕ

- Для объектов предметной области отделяйте модели записи от моделей чтения. Клиентам, которые заинтересованы только в получении данных от сущности, следует использовать специальный объект вместо сущности, предоставляющей методы для изменения ее состояния.
- Модель чтения можно создать непосредственно из модели записи, но более эффективный способ — создавать ее из источника данных, используемого моделью записи. Если это невозможно или неэффективно, используйте события предметной области для создания модели чтения с течением времени.

ОТВЕТЫ К УПРАЖНЕНИЯМ

1 Правильный ответ: **б**. Код действительно извлекает информацию из модели (он вызывает метод `wasCancelled()`), но он также изменяет объект (вызывает его метод `finalize()`). Это делает его моделью записи, поскольку модель чтения не предлагает методы для изменения состояния модели обычным клиентам.

2 Правильный ответ: **а**. Теоретически это может быть моделью записи, но на самом деле так быть не должно. Ее единственная цель — показать информацию о митапе путем предоставления пользователю HTML-ответа.

3 Возможный ответ: вам понадобятся события, представляющие все, что может произойти с корзиной покупок. Учитывая описанные функции, этими событиями могут быть `ProductWasAddedToCart`, `ProductWasRemovedFromCart` и `ProductQuantityWasModified`. Совет, как и всегда, — искать условия, относящиеся к конкретной предметной области, и использовать слова, которые употребляют отраслевые эксперты. Почему нет события с названием `CartWasCreated`? Потому что факт создания корзины уже подразумевается в `ProductWasAddedToCart`.

9 Изменение поведения сервисов

В этой главе:

- ✓ Изменение поведения без изменения кода
- ✓ Создание настраиваемых и заменяемых поведений
- ✓ Введение абстракций для композиции и декорирования
- ✓ Избегание наследования для переопределения поведения объектов
- ✓ Создание классов `final` и методов `private` для предотвращения злоупотреблений объектами

Можно проектировать сервисы так, чтобы они создавались и использовались определенным образом. Но природа программного проекта такова, что со временем он будет меняться. Вам придется часто изменять класс, чтобы при использовании он вел себя желаемым образом. Однако модификация класса сопряжена с определенными издержками: существует опасность нарушить его работу. Распространенной альтернативой изменениям в классе выступает переопределение ряда его методов, но это способно вызвать еще больше проблем. Вот почему в целом предпочтительнее модифицировать структуру объектного графа вместо кода в классе. Лучше заменять детали, чем изменять их.

9.1. ВВЕДИТЕ АРГУМЕНТЫ КОНСТРУКТОРА, ЧТОБЫ СДЕЛАТЬ ПОВЕДЕНИЕ НАСТРАИВАЕМЫМ

Мы уже обсуждали, что сервисы должны создаваться за один раз со всеми зависимостями и значениями конфигурации, предоставляемыми в качестве аргументов конструктора. Если необходимо изменить поведение объекта, в дело снова вступает конструктор. Если вы хотите повлиять на поведение сервиса, всегда лучше использовать заменяемый аргумент конструктора.

Возьмем, к примеру, следующий класс, `FileLogger`, который записывает сообщения в файл.

Листинг 9.1. Класс FileLogger

```
final class FileLogger
{
    public function log(message): void
    {
        file_put_contents(
            '/var/log/app.log',
            message,
            FILE_APPEND
        );
    }
}
```

Чтобы перенастроить журнал для записи сообщений в другой файл, укажите путь к файлу журнала в качестве аргумента конструктора, который копируется в свойство.

Листинг 9.2. Используйте аргумент конструктора для настройки FileLogger

```
final class FileLogger
{
    private string filePath;

    public function __construct(string filePath)
    {
        this.filePath = filePath;
    }

    public function log(message): void
    {
        file_put_contents(this.filePath, message, FILE_APPEND);
    }
}

logger = new FileLogger('/var/log/app.log');
```

УПРАЖНЕНИЕ

1 Выберите оптимальный вариант, как сделать базовый URL-адрес следующего клиента API настраиваемым.

```
final class ApiClient
{
    public function sendRequest(
        string method,
        string path
    ): Response {
        url = 'https://api.acme.com' . path;

        // ...

    }
}
```

а Добавить объект `Config` в качестве аргумента конструктора, из которого `ApiClient` может получить базовый URL-адрес.

б Добавить базовый URL-адрес в виде строки или объекта-значения в конструктор `Api-Client`.

в Добавить `baseUrl` в качестве дополнительного параметра в `sendRequest()`.

9.2. ВВЕДИТЕ АРГУМЕНТЫ КОНСТРУКТОРА, ЧТОБЫ СДЕЛАТЬ ПОВЕДЕНИЕ ЗАМЕНЯЕМЫМ

Мы уже знаем, что каждая зависимость сервиса должна вводиться в качестве аргумента конструктора. Аналогично тому как можно менять значения конфигурации, можно заменить и зависимости.

Рассмотрим `ParameterLoader`, который используется для загрузки списка ключей и значений («параметров») из файла JSON.

Листинг 9.3. Класс ParameterLoader

```
final class ParameterLoader
{
    public function load(filePath): array
    {
        rawParameters = json_decode(
            file_get_contents(filePath),
            true
        );
```

Загрузите параметры из файла и добавьте их к уже загруженным

```
        parameters = [];

        foreach (rawParameters as key => value) {
            parameters[] = new Parameter(key, value);
        }
        return parameters;
    }
}

loader = new ServiceConfigurationLoader(
    __DIR__ . '/parameters.json'
);
```

Какую часть этого класса следует заменить, чтобы добавить поддержку загрузки XML или, возможно, даже файла YAML? Большая часть `Parameterloader` довольно универсальна, за исключением вызова `json_decode()`. Чтобы сделать данную часть заменяемой, придется ввести абстракцию. Это означает поиск более абстрактной концепции, нежели декодирование файла JSON, и добавление интерфейса, который может представлять абстракцию.

Абстрактное понятие — «загрузка файла», поэтому `FileLoader` может стать подходящим именем интерфейса, который представляет эту задачу в коде. У нас будет стандартная реализация для этого интерфейса, загружающего параметры из файла JSON. Назовем ее `JsonFileLoader`.

Листинг 9.4. Интерфейс FileLoader, реализованный с помощью JsonFileLoader

```
interface FileLoader
{
    public function loadFile(string filePath): array
}

final class JsonFileLoader implements FileLoader
{
    public function loadFile(string filePath): array
    {
        Assertion.isFile(filePath);

        result = json_decode(
            file_get_contents(filePath),
            true
        );

    if (!is_array(result)) {
        throw new RuntimeException(
            'Декодирование "{filePath}" не привело к массиву'
            );
        }

        return result;
    }
}
```

Загрузите массив пар ключ/значение, представляющих параметры, хранящиеся в файле в заданном местоположении

Мы воспользовались возможностью добавить ряд проверок предусловий и постусловий, чтобы сделать реализацию, специфичную для JSON, более надежной.

Теперь нужно убедиться, что `parameterloader` получает экземпляр `FileLoader`, введенный в качестве аргумента конструктора, и мы заменим существующий код загрузки файла в `parameterloader` вызовом `FileLoader.LoadFile()`.

Листинг 9.5. Parameterloader зависит от экземпляра FileLoader

```
final class ParameterLoader
{
    private FileLoader fileLoader;

    public function __construct(FileLoader fileLoader)
    {
        this.fileLoader = fileLoader;
    }

    public function load(filePath): array
    {
        // ...

        foreach (/* ... */) {
            if (/* ... */) {
                rawParameters = this.fileLoader.loadFile(
                    filePath
                );
            }
        }

        // ...
    }
}

parameterLoader = new ParameterLoader(new JsonFileLoader());
parameterLoader.load(__DIR__ . '/parameters.json');
```

Абстрагировав часть поведения `ParameterLoader`, можно заменить его любой другой *конкретной* реализацией, такой как загрузчик файлов XML или YAML.

Листинг 9.6. Заменить реализацию FileLoader несложно

```
final class XmlFileLoader implements FileLoader
{
    // ...
}

parameterLoader = new ParameterLoader(new XmlFileLoader());
parameterLoader.load(__DIR__ . '/parameters.xml');
```

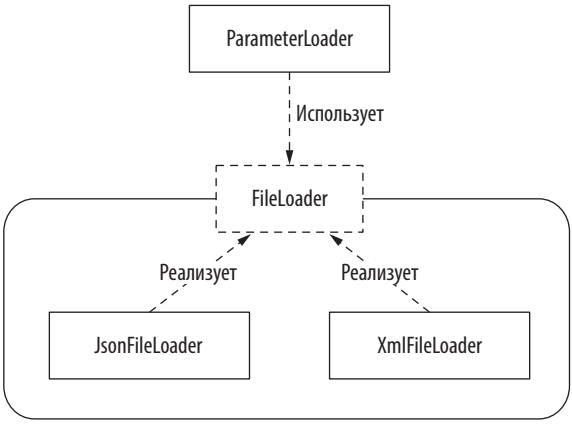


Рис. 9.1. До тех пор пока зависимость FileLoader от Parameterloader следует соглашению, определяемому интерфейсом, не имеет значения, что происходит за кулисами реального добавляемого загрузчика файлов

УПРАЖНЕНИЕ

2 Сделайте поведение *форматирования* и *записи* следующего класса `Logger` заменяемым.

```
final class Logger
{
    private string logFilePath;

    public function __construct(string logFilePath)
    {
        this.logFilePath = logFilePath;
    }

    public function log(string message, array context): void
    {
        handle = fopen(logFilePath);

        fwrite(
            handle,
            message . ' ' . json_encode(context)
        );
    }
}
```

9.3. СОЗДАВАЙТЕ АБСТРАКЦИИ, ЧТОБЫ ДОБИТЬСЯ БОЛЕЕ СЛОЖНОГО ПОВЕДЕНИЯ

При наличии надлежащей абстракции будет легко скомпоновать несколько конкретных экземпляров в более сложное поведение. Например, что делать, если

необходимо поддерживать несколько форматов на основе расширения имени файла? Этого можно добиться, используя композицию объекта следующим образом.

Листинг 9.7. MultipleLoaders — это FileLoader, выступающий оберткой для других FileLoader

```
interface FileLoader
{
    /**
     * ...
     *
     * @throws CouldNotLoadFile
     */
    public function loadFile(string filePath): array
}

final class MultipleLoaders implements FileLoader
{
    private array loaders;

    public function __construct(array loaders)
    {
        Assertion.allIsInstanceOf(loaders, FileLoader.className);
        this.loaders = loaders;
    }

    public function loadFile(string filePath): array
    {
        lastException = null;

        foreach (this.loaders as loader) {
            try {
                return loader.loadFile(filePath);
            } catch (CouldNotLoadFile exception) {
                lastException = exception;
            }
        }

        throw new CouldNotLoadFile(
            'Ни один из загрузчиков файлов не смог загрузить файл "{filePath}"',
            LastException
        );
    }
}
```

Добавьте аннотацию к интерфейсу, указав на то, что загрузка файла может выдать исключение CouldNotLoadFile

Добавьте новый загрузчик файлов, состоящий из нескольких экземпляров FileLoader. Когда его просят загрузить файл, он делегирует вызов загрузчикам до тех пор, пока один из них не выдаст CouldNotLoadFile

Обратите внимание, что новая логика размещена за пределами самого `Parameterloader`, который не знает, что происходит за используемым им интерфейсом `FileLoader`.

Вместо того чтобы просто пробовать разные загрузчики, можно применить несколько иной подход. Например, каждый загрузчик может быть зарегистрирован для определенного расширения файла.

В следующем листинге показано, как это сделать (опять же используя композицию объектов).

Листинг 9.8. Альтернативная реализация MultipleLoaders

```
final class MultipleLoaders implements FileLoader
{
    private array loaders;

    public function __construct(array loaders)
    {
        Assertion.allIsInstanceOf(loaders, FileLoader.className);
        Assertion.allIsString(array_keys(loaders));
        this.loaders = loaders;
    }

    public function loadFile(string filePath): array
    {
        extension = pathinfo(filePath, PATHINFO_EXTENSION);
        if (!isset(this.loaders[extension])) {
            throw new CouldNotLoadFile(
                'Нет загрузчика для расширения файла "{extension}"'
            );
        }

        return this.loaders[extension].loadFile(filePath);
    }
}

parameterLoader = new ParameterLoader(
    new MultipleLoaders([
        'json' => new JsonFileLoader(),
        'xml' => new XmlFileLoader()
    ]);
);
parameterLoader.load('parameters.json');
parameterLoader.load('parameters.xml');

parameterLoader.load('parameters.yml');
```

Предполагается, что this.loaders — это карта ключей и значений, где ключ — расширение файла, а значение — FileLoader, который следует использовать для загрузки файла с этим расширением

Это вызовет исключение CouldNotLoadFile

Как видим, эта настройка очень динамична. Однако всегда имейте в виду, что при написании такого кода для проекта обычно не нужно поддерживать все эти форматы файлов. Введение абстракции `FileLoader` — разумный ход, но написание различных реализаций загрузчика следует рассматривать как «обобщение до того, как оно потребуется». *Пока не потребуется...*

9.4. ДЕКОРИРУЙТЕ СУЩЕСТВУЮЩЕЕ ПОВЕДЕНИЕ

В предыдущем примере все загрузчики файлов для JSON, XML и т. д. возвращают массив необработанных параметров (пар «ключ — значение»). Что, если

нам понадобится разрешить пользователю использовать переменные среды в качестве значений для этих параметров? Копировать эту логику замены во все реализации `FileLoader` — не самое удачное решение. Вместо этого можно добавить поведение поверх любого существующего поведения. Это можно сделать, используя особый стиль композиции, называемый *декорированием*, как показано в следующем листинге.

Листинг 9.9. ReplaceParametersWithEnvironmentVariables

```
final class ReplaceParametersWithEnvironmentVariables
    implements FileLoader
{
    private FileLoader fileLoader;
    private array envVariables;

    public function __construct(
        FileLoader fileLoader,
        array envVariables
    ) {
        this.fileLoader = fileLoader;
        this.envVariables = envVariables;
    }

    public function loadFile(string filePath): array
    {
        parameters = this.fileLoader.loadFile(filePath);

        foreach (parameters as key => value) {
            parameters[key] = this.replaceWithEnvVariable(
                value
            );
        }

        return parameters;
    }

    private function replaceWithEnvVariable(string value): string
    {
        if (isset(this.envVariables[value])) {
            return this.envVariables[value];
        }

        return value;
    }
}

parameterLoader = new ParameterLoader(
    new ReplaceParametersWithEnvironmentVariables(
        new MultipleLoaders([
            'json' => new JsonFileLoader(),
            'xml' => new XmlFileLoader()
```

Реальный загрузчик файлов вводится в качестве аргумента конструктора

Мы используем реальный загрузчик файлов для загрузки файла

Любое значение параметра, которое также является именем переменной среды, будет заменено значением последней

```
        ]),
        [
            'APP_ENV' => 'dev',
        ]
    )
);
```

Декорирование также часто используется, когда стоимость использования реального сервиса довольно высока. Например, если приложению приходится загружать и анализировать файл parameters.json много раз, разумнее будет обернуть исходный сервис и запомнить последний результат, который он вернул.

Листинг 9.10. CachedFileLoader вызывает реальный загрузчик только в случае необходимости

```
final class CachedFileLoader implements FileLoader
{
    private FileLoader realLoader;

    private cache = [];

    public function __construct(FileLoader realLoader)
    {
        this.realLoader = realLoader;
    }

    public function loadFile(string filePath): array
    {
        if (isset(this.cache[filePath])) {
            return this.cache[filePath];
        }

        result = this.realLoader.loadFile(filePath);

        this.cache[filePath] = result;

        return result;
    }
}

loader = new CachedFileLoader(new JsonFileLoader());

loader.load('parameters.json');

loader.load('parameters.json');
```

Мы уже загружали этот файл раньше, поэтому можем вернуть кэшированный результат

Мы еще не загружали этот файл, так что сделаем это сейчас

Мы храним результат в кэше, поэтому не придется загружать файл заново в следующий раз

Это позволит передать вызов в JsonFileLoader

Во второй раз мы не будем обращаться к файловой системе

Преимущество использования композиции в этом сценарии заключается в том, что логика кэширования не должна дублироваться в разных реализациях загрузчика файлов. Фактически логика в `CachedFileLoader` не зависит от используемой реализации `FileLoader`. Это означает, что ее можно и тестировать, и разрабатывать отдельно. Если потребуется сделать логику кэширования более продвинутой,

потребуется всего лишь изменить код этого единственного класса, предназначенного для кэширования.

УПРАЖНЕНИЕ

3 Все операторы `log()` в следующем коде отвлекают от реальной цели класса — импорта CSV-файла. Используйте декорирование и композицию, чтобы переместить операторы `log()` в отдельные классы. (Подсказка: чтобы осуществить декорирование, потребуется ввести специальный объект для импорта одной строки.)

```
final class CsvFileImporter
{
    private Logger logger;

    public function __construct(Logger logger)
    {
        this.logger = logger;
    }

    public function import(string csvFile): void
    {
        this.logger.log('Importing file: ' . csvFile);

        foreach (linesIn(csvFile) as lineNumber => line) {
            this.logger.log('Importing line: ' . lineNumber);

            // import the line
            fields = fieldsIn(line);
            // ...

            this.logger.log('Imported line: ' . lineNumber);
        }

        this.logger.log('Finished importing');
    }
}
```

9.5. ИСПОЛЬЗУЙТЕ ОБЪЕКТЫ УВЕДОМЛЕНИЙ ИЛИ ПРОСЛУШИВАТЕЛИ СОБЫТИЙ ДЛЯ ДОБАВЛЕНИЯ ПОВЕДЕНИЯ

Мы уже рассматривали использование прослушивателей в качестве способа отделения основного командного метода от второстепенных задач. Этот способ можно использовать, если вы хотите перенастроить сервисы для выполнения других задач. В качестве примера рассмотрим сервис `ChangeUserPassword`.

Листинг 9.11. Сервис ChangeUserPassword

```
final class ChangeUserPassword
{
    private PasswordEncoder passwordEncoder;

    public function __construct(
        PasswordEncoder passwordEncoder,
        /* ... */
    ) {
        // ...
    }

    public function changeUserPassword(
        UserId userId,
        string plainTextPassword
    ): void {
        encodedPassword = this.passwordEncoder.encode(
            plainTextPassword
        );

        // Store the new password...
    }
}
```

Новое требование к сервису заключается в том, что он должен отправлять электронное письмо пользователю после регистрации с уведомлением, что его пароль изменился (ведь это мог сделать хакер!). Вместо добавления кода к существующему классу и методу лучше отправить событие и настроить прослушиватель, который и будет отправлять электронное письмо.

Листинг 9.12. Класс события UserPasswordChanged и его прослушиватель

```
final class UserPasswordChanged
{
    private UserId userId;

    public function __construct(UserId userId)
    {
        this.userId = userId;
    }
}

final class SendUserPasswordChangedNotification
{
    // ...

    public function whenUserPasswordChanged(
        UserPasswordChanged event
    ): void {
        // Отправить письмо...
    }
}
```

Определяем новый тип события

Определяем прослушиватель для этого события

Наконец, нужно переписать сервис `ChangeUserPassword` для отправки только что определенного события `UserPasswordChanged`.

Листинг 9.13. ChangeUserPassword отправляет событие UserPasswordChanged

```
final class ChangeUserPassword
{
    private EventDispatcher eventDispatcher;

    public function __construct(
        /* ... */,
        EventDispatcher eventDispatcher
    ) {
        // ...
    }

    public function changeUserPassword(
        UserId userId,
        string plainTextPassword
    ): void {
        encodedPassword = this.passwordEncoder.encode(
            newPassword
        );

        // Сохранить новый пароль

        this.eventDispatcher.dispatch(
            new UserPasswordChanged(userId)
        );
    }
}

listener = new SendUserPasswordChangedNotification(/* ... */);
eventDispatcher = new EventDispatcher([
    UserPasswordChanged.className => [
        listener,
        'whenUserPasswordChanged'
    ]
]);

service = new ChangeUserPassword(/* ... */, eventDispatcher);

service.changeUserPassword(new UserId(/* ... */), 'Test123');
```

Необходимо убедиться, что прослушиватель зарегистрирован правильно

Событие UserPasswordChanged будет отправлено прослушивателю SendUserPasswordChangedNotification

Преимущество использования диспетчера событий в том, что он позволяет добавлять новое поведение в сервис без изменения существующей логики. Добавленный диспетчер событий дает возможность добавить и новое поведение. Всегда можно зарегистрировать новый прослушиватель для существующего события.

Недостаток использования диспетчера событий в том, что у него слишком обобщенное имя. При чтении кода не совсем понятно, что происходит за вызовом `dispatch()`. Кроме того, довольно сложно определить, какие прослушиватели

должны реагировать на то или иное событие. Альтернативное решение — добавить собственную абстракцию.

В качестве примера возьмем следующий класс `Importer`, который импортирует CSV-файлы из заданного каталога и отправляет события, позволяющие другим сервисам прослушивать процесс импорта.

Листинг 9.14. Importer отправляет события

```
final class Importer
{
    private EventDispatcher dispatcher;

    public function __construct(EventDispatcher dispatcher)
    {
        this.dispatcher = dispatcher;
    }

    public function import(string csvDirectory): void
    {
        foreach (Finder.in(csvDirectory).files() as file) {
            Read the file
            lines = /* ... */;

            foreach (lines as index => line) {
                if (index == 0) {
                    // Parse the header
                    header = /* ... */;

                    this.dispatcher.dispatch(
                        new HeaderImported(file, header)
                    );
                }
                else {
                    data = /* ... */;

                    this.dispatcher.dispatch(
                        new LineImported(file, index)
                    );
                }
            }

            this.dispatcher.dispatch(
                new FileImported(file)
            );
        }
    }
}
```

Оказывается, у каждого из этих событий есть только один прослушиватель — тот, который записывает отладочную информацию о событии в файл журнала. Хотя это очень простая задача, для нее требуется много кода: нужно написать

классы события и прослушивателя, не забывая о правильной регистрации прослушивателей.

Как вы теперь знаете, большинство таких прослушивателей выполняют одну и ту же работу, поэтому вместо того чтобы распространять это поведение на несколько классов, можно объединить его в одном и ввести для него собственную абстракцию: `ImportNotifications`.

Листинг 9.15. Одна абстракция может заменить все события импорта

```
interface ImportNotifications
{
    public function whenHeaderImported(
        string file,
        array header
    ): void;

    public function whenLineImported(
        string file,
        int index
    ): void;

    public function whenFileImported(
        string file
    ): void;
}

final class ImportLogging implements ImportNotifications
{
    private Logger logger;

    public function __construct(Logger logger)
    {
        this.logger = logger;
    }

    public function whenHeaderImported(
        string file,
        array header
    ): void {
        this.logger.debug('Imported header ...');
    }

    // И так далее...

}
```

Вместо того чтобы вводить диспетчер событий в класс `Importer`, можно добавить экземпляр `ImportNotifications` и вместо вызова `dispatch()` вызывать выделенный метод события для добавленного экземпляра `ImportNotifications`.

Листинг 9.16. Importer вызывает Importnotifications вместо EventDispatcher

```
final class Importer
{
    private ImportNotifications notify;

    public function __construct(ImportNotifications notify)
    {
        this.notify = notify;
    }

    public function import(string csvDirectory): void
    {
        foreach (Finder.in(csvDirectory).files() as file) {
            // Read the file
            lines = /* ... */;

        foreach (lines as index => line) {
            if (index == 0) {
            // Parse the header
            header = /* ... */;
                    this.notify.whenHeaderImported(
                        file,
                        header
                    )
                }
                else {
                    data = /* ... */;

                    this.notify.whenLineImported(file, index);
                }
            }

            this.notify.whenFileImported(file);
        }
    }
}
```

Если помимо ведения журнала вам понадобится вывести отладочную информацию на экран, это можно легко сделать в том же классе. А можно добавить другой класс и снова использовать композицию для вызова обоих поведений вместо одного.

9.6. НЕ ИСПОЛЬЗУЙТЕ НАСЛЕДОВАНИЕ ДЛЯ ИЗМЕНЕНИЯ ПОВЕДЕНИЯ ОБЪЕКТА

Взглянем еще раз на `ParameterLoader`, который мы обсуждали выше. Что делать, если исходный класс выглядит как в следующем листинге?

Листинг 9.17. Другой вариант ParameterLoader

```
class ParameterLoader
{
    public function load(filePath): array
    {
        // ...

        rawParameters = this.loadFile(filePath);

        // ...

        return parameters;
    }

    protected function loadFile(string filePath): array
    {
        return json_decode(
            file_get_contents(filePath),
            true
        );
    }
}
```

В этом варианте два ключевых изменения:

- Класс `ParameterLoader` не помечен как `final`, а это означает, что можно определить подкласс, который его расширяет.
- Имеется специальный метод для загрузки файла, и этот метод помечен как `protected`, это означает, что он может быть переопределен таким подклассом.

Теперь, когда внутренние компоненты класса полностью открыты, можно расширить класс, наследовать основную логику и переопределить часть загрузки файлов, чтобы она работала с XML.

Листинг 9.18. Загрузка XML-файлов возможна, если расширить ParameterLoader

```
final class XmlFileParameterLoader extends ParameterLoader
{
    protected function loadFile(string filePath): array
    {
        rawXml = file_get_contents(filePath);

        // ...

        return /* ... */;
    }
}
```

Вполне очевидно, что это решение не обладает всеми преимуществами предыдущего, такими как абстракция загрузчика файлов, которая предлагала чистые

варианты компоновки, поддержку нескольких загрузчиков файлов одновременно и т. д. Это альтернативное решение, в котором мы расширяем существующий класс `ParameterLoader`, не обладает той гибкостью и возможностью перенастройки. Использование наследования классов для изменения поведения существующего объекта имеет недостатки:

- *Подкласс и родительский класс становятся связанными* — изменение деталей реализации, которые обычно скрыты за общедоступным интерфейсом класса, теперь может привести к нарушению реализации подкласса. Подумайте, что произойдет, если имя этого `protected`-метода будет изменено или если он получит дополнительный обязательный параметр.
- *Подклассы могут переопределять методы как* `protected`, *так и* `public` — подклассы получают доступ к `protected`-свойствам и их типам данных, которые до сих пор были внутренней информацией. Другими словами, многие внутренние части объекта становятся доступны.

Что, если вместо этого родительский класс предложит так называемый *шаблонный* метод и позволит разработчику предоставлять только его, не раскрывая больше внутренних компонентов, чем это необходимо? В следующем листинге показано, как это будет выглядеть.

Листинг 9.19. ParameterLoader, реализующий шаблонный метод

```
abstract class ParameterLoader
{
    // ...

    final public function load(filePath): array
    {
        parameters = [];

        foreach (/* ... */) {
            // ...
            if (/* ... */) {
                rawParameters = this.loadFile(filePath);
                // ...
            }
        }

        return parameters;
    }

    abstract protected function loadFile(string filePath): array;
}
```

Отметьте все свойства как private, чтобы они оставались закрытыми для родительского класса. Отметьте все методы как final, чтобы их невозможно было переопределить

Разрешаем реализацию (не переопределение) только для одного метода

Уже лучше, но все равно не оптимально. Возможно, мы избавились от недостатков, вызванных наследованием, но у нас не стало возможности неограниченного использования композиции.

Основываясь на этом примере, можно сделать вывод, что все, что можно реализовать с помощью шаблонного метода, можно реализовать и с помощью композиции. Единственное, что нужно сделать, — преобразовать `abstract protected` метод в обычный метод `public` для добавленного объекта. После этого можно снова объявить сам класс `final`. В случае `ParameterLoader` мы уже это сделали.

Листинг 9.20. ParameterLoader помечен как final

```
final class ParameterLoader
{
    private FileLoader fileLoader;

    public function __construct(FileLoader fileLoader)
    {
        this.fileLoader = fileLoader;
    }

    final public function load(filePath): array
    {
        parameters = [];

        foreach (/* ... */) {
            // ...
            if (/* ... */) {
                rawParameters = this.fileLoader.loadFile(
                    filePath
                );

                // ...
            }
        }

        return parameters;
    }
}
```

Используем публичный метод LoadFile() из введенного загрузчика файлов вместо protected LoadFile(), который мы использовали ранее

Поскольку многие проекты по-прежнему не помечают классы как `final` по умолчанию, вы столкнетесь со многими фреймворками и библиотеками, которые позволяют изменять поведение объектов путем расширения их классов. Пожалуйста, не делайте этого. Всегда выбирайте решение, которое использует только `public`-методы, и лучше те, которые являются частью опубликованного интерфейса класса. Не полагайтесь на внутренние компоненты класса, наследуя их от другого класса. Это сделает решения более хрупкими, потому что в их основе будут процессы, которые с большой вероятностью изменят опубликованный, поддерживаемый API, предлагаемый платформой или библиотекой.

9.6.1. Когда можно использовать наследование?

В целом наследование стоит использовать только для задания строгой иерархии типов. Например, блок содержимого может быть абзацем или изображением,

и тогда вы напишете: `Paragraph extends ContentBlock` и `Image extends ContentBlock`. На практике я редко нахожу веские аргументы в пользу наследования. Обычно оно выходит криво или «вынужденно» и вскоре начинает мешать.

Наследование обычно применяется для повторного использования кода, но композиция в этом случае гораздо эффективнее. Однако некоторые типы объектов, такие как сущности или объекты-значения, не поддерживают внедрение зависимостей, поэтому невозможно добиться повторного использования кода таким образом. В этом случае я рекомендую использовать трактовки (traits). Трактовки не являются наследованием, потому что имя трактовки в конечном итоге не становится частью иерархии класса, как это было бы с родительским классом или интерфейсом. Трактовка — это повторное использование простого кода, то есть копирование/вставка на уровне компилятора.

Если, например, вы хотите записывать события во всех сущностях, определите для них следующий интерфейс. Это позволит убедиться, что у всех сущностей есть методы для получения этих событий и очистки после отправки.

```
interface RecordsEvents
{
    public function releaseEvents(): array;

    public function clearEvents(): void;
}
```

Поскольку все сущности будут иметь одинаковую реализацию для этих методов и вы не хотите вручную копировать/вставлять эту реализацию во все классы сущностей, вполне можно использовать `trait`:

```
trait EventRecordingCapabilities
{
    private array events;

    private function recordThat(object event): void
    {
        this.events[] = event;
    }

    public function releaseEvents(): array
    {
        return this.events;
    }

    public function clearEvents(): void
    {
        this.events = [];
    }
}
```

Сущности должны лишь реализовать этот интерфейс и использовать сопутствующую трактовку, и у них появятся возможности записи событий:

```
final class Product implements RecordsEvents
{
    use EventRecordingCapabilities;

    // ...
}
```

9.7. ПОМЕЧАЙТЕ КЛАССЫ КАК FINAL ПО УМОЛЧАНИЮ

Что касается сервисов, мы уже обосновали необходимость пометки классов как `final`: изменение поведения с помощью композиции объектов вместо наследования — лучший и более гибкий способ. Если вы пойдете этим путем, вам вообще не понадобится расширение класса. Такие объекты способны хранить внутренние данные при себе и разрешать клиентам использовать поведение, которое является частью их общедоступного интерфейса. Это означает, что каждый класс может и должен быть помечен как `final`. Так клиент понимает, что класс не предназначен для расширения, а его методы не предназначены для переопределения. Это же заставит пользователей искать лучшие способы изменить его поведение.

В отношении других типов объектов, таких как сущности и объекты-значения, также следует задать вопрос: должны ли и они стать `final`? Да, должны. Эти объекты представляют понятия предметной области и знания, которые вы получили о них. Было бы странно переопределять часть поведения этих классов, расширяя их. Если вы узнали о предметной области то, что заставляет вас изменить поведение объекта, не создавайте для этого подкласс, а измените саму сущность.

Единственное исключение из этого правила — задание иерархии объектов. В таком случае расширение родительского класса может указывать на связь между этими объектами: подкласс следует рассматривать как частный случай родительского класса. Тогда родительский класс не будет `final`, потому что подкласс должен иметь возможность расширять его.

9.8. ПОМЕЧАЙТЕ МЕТОДЫ И СВОЙСТВА КАК PRIVATE ПО УМОЛЧАНИЮ

До сих пор все примеры в этой книге содержали классы `final` со свойствами `private`. Как только вы сделаете классы `final`, вы заметите, что больше нет необходимости иметь свойства `protected`. Классы, как правило, не будут использоваться

для расширения, поэтому все их внутренние компоненты останутся при них. Единственный способ, которым клиенты могут взаимодействовать с объектом, — это создать его и вызвать для него общедоступные методы. Закрыв само определение класса, можно создать несколько действительно сильных объектов. Свобода изменять внутренние элементы объекта, если это не нарушает соглашения, определенного его опубликованным интерфейсом, — это большое преимущество.

УПРАЖНЕНИЯ

4 Для следующего класса не существует классов, которые его расширяют, и он не был разработан для расширения. Что необходимо изменить в определении класса?

```
class Product
{
    protected int $id;
    protected string description;

    // ...
}
```

а Класс должен быть помечен как abstract.

б Класс должен быть помечен как final.

в Свойства должны быть помечены как private.

г Свойства должны быть помечены как public.

5 Что не так со следующим кодом?

```
class Preferences
{
    private string preferencesFilePath;

    public function __construct(string preferencesFilePath)
    {
        this.preferencesFilePath = preferencesFilePath;
    }

    public function getPreference(
        string preference,
        bool defaultValue
    ): bool {
        preferences = this.loadPreferences();

        if (isset(preferences[preference])) {
            return preferences[preference];
        }
```

```
            return defaultValue;
        }

        protected function loadPreferences(): array
        {
            return json_decode(
                file_get_contents(preferencesFilePath)
            );
        }
    }

    final class DatabaseTablePreferences extends Preferences
    {
        private Connection connection;

        public function __construct(Connection connection)
        {
            this.connection = connection;
        }

        protected function loadPreferences(): array
        {
            return this.connection.executeQuery(
                'SELECT * FROM preferences'
            ).fetchAll();
        }
    }
```

а `DatabaseTablePreferences` использует наследование для изменения поведения сервиса.

б Вместо этого `Preferences` должен расширять `DatabaseTablePreferences`.

в `Preferences` должен отправлять события, позволяющие загружать настройки из другого местоположения.

г Загрузка настроек должна быть делегирована выделенному сервису с собственным интерфейсом.

ЗАКЛЮЧЕНИЕ

- Если необходимо изменить поведение сервиса, ищите способы настроить это поведение с помощью аргументов конструктора. Если такой способ не подходит, потому что придется менять значительную часть логики, поищите способы замены зависимостей, которые также передаются в качестве аргументов конструктора.

- Если поведение, которое вы хотите изменить, еще не представлено зависимостью, извлеките его, введя абстракцию: концепцию более высокого уровня и интерфейс. Тогда вы получите часть, которую можно заменить, а не модифицировать. Абстракция предлагает возможность создавать и декорировать модели поведения, чтобы усложнять их без информирования (или изменения) исходного сервиса.
- Не применяйте наследование для изменения поведения сервиса путем переопределения его методов. Всегда ищите решения, использующие композицию объектов. А лучше полностью закройте все классы для наследования: пометьте их как `final` и сделайте все свойства и методы `private`, если они не являются частью открытого интерфейса класса.

ОТВЕТЫ К УПРАЖНЕНИЯМ

1 Правильный ответ: **б**. Внедрение универсального объекта `Config` — не очень хорошая идея (см. также раздел 2.3). Гораздо больше смысла имеет ввод значения конфигурации специально для базового URL-адреса. И хотя передача его в качестве аргумента в `sendRequest()` — один из возможных вариантов, это приведет к тому, что все клиенты будут знать это значение. И это не только плохо для работоспособности кода, но и очень неудобно для клиентов.

2 Возможный ответ: форматирование сообщений журнала и запись сообщений журнала должны иметь собственные интерфейсы и стандартные реализации, основанные на коде, который уже есть в `Logger`:

```
interface Formatter
{
    public function format(
        string message,
        array context
    ): string;
}

final class JsonEncodedContextFormatter implements Formatter
{
    public function format(string message, array context)
    {
        return message . ' ' . json_encode(context);
    }
}

interface Writer
{
    public function write(string formattedMessage): void;
}
```

```
final class FileWriter implements Writer
{
    public function write(string formattedMessage): void
    {
        handle = fopen(logFilePath);
        fwrite(formattedMessage);
    }
}

final class Logger
{
    private Formatter formatter;
    private Writer writer;
    public function __construct(
        Formatter formatter,
        Writer writer
    ) {
        this.formatter = formatter;
        this.writer = writer;
    }

    public function log(string message, array context): void
    {
        this.writer.write(
            this.formatter.format(message, context)
        );
    }
}
```

3 Возможный ответ: следующий код показывает один из вариантов решения. Поскольку много кода требуется лишь для того, чтобы избавиться от операторов ведения журнала, можно обратиться к решению из аспектно-ориентированного программирования (AOP). Инструменты AOP позволяют подключаться к существующим вызовам методов и запускать код до или после исходного метода.

```
interface LineImporter
{
    public function import(int lineNumber, string line): void;
}

final class DefaultLineImporter implements LineImporter
{
    public function import(int lineNumber, string line): void
    {
        // import the line
        fields = fieldsIn(line);
        // ...
    }
}
```

Интерфейс LineImporter определяет точку расширения для импорта строки

DefaultLineImporter содержит исходный код для импорта строки

```
final class LoggingLineImporter implements LineImporter
{
    private LineImporter actualLineImporter
    private Logger logger;

    public function __construct(
        LineImporter actualLineImporter
        Logger logger
    ) {
        this.actualLineImporter = actualLineImporter;
        this.logger = logger;
    }

    public function import(int lineNumber, string line): void
    {
        this.logger.log('Importing line: ' . lineNumber);

        this.actualLineImporter.import(lineNumber, line);

        this.logger.log('Imported line: ' . lineNumber);
    }
}

interface FileImporter
{
    public function import(string file): void;
}

final class CsvFileImporter implements FileImporter
{
    private LineImporter lineImporter

    public function __construct(LineImporter lineImporter)
    {
        this.lineImporter = lineImporter;
    }

    public function import(string file): void
    {
        foreach (linesIn(csvFile) as lineNumber => line) {
            this.lineImporter.import(lineNumber, line);
        }
    }
}

final class LoggingFileImporter реализует FileImporter
{
    private Logger logger;
    private FileImporter actualFileImporter

    public function __construct(
        FileImporter actualFileImporter,
        Logger logger
```

LoggingLineImporter добавляет запись в журнал до и после фактического импорта строки

Интерфейс FileImporter позволяет декорировать исходный CsvFileImporter

Оригинальный CsvFileImporter реализует FileImporter

LoggingFileImporter добавляет ведение журнала до и после импорта файла

```
    ) {
        this.actualFileImporter = actualFileImporter;
        this.logger = logger;
    }

    public function import(string csvFile): void
    {
        this.logger.log('Importing file: ' . csvFile);

        this.actualFileImporter.import(csvFile);

        this.logger.log('Finished importing');
    }
}

logger = // ...

importer = new LoggingFileImporter(
    new CsvFileImporter(
        new LoggingLineImporter(
            new DefaultLineImporter(),
            logger
        )
    ),
    logger
);
importer.import(/* ... */);
```

Создание экземпляра импортера стало сложнее, но его использование не изменилось

4 Правильные ответы: **б** и **в**. Пометка класса `abstract` будет означать нечто противоположное, а именно, что он предназначен для расширения. Пометка свойств как `public` полностью раскрывает их клиентам `Product`, что почти никогда не бывает желаемым.

5 Правильные ответы: **а** и **г**. Использовать наследование для изменения поведения сервиса не рекомендуется. Расширение никак не улучшает ситуацию, равно как и использование событий. Событие должно быть уведомлением, позволяющим другим сервисам предпринимать дальнейшие действия, а не изменять поведение самого сервиса. Вместо использования наследования класс `Preference` должен переносить загрузку настроек в другой сервис — тот, что можно заменить и/или декорировать.

10 Справочник объектов

В этой главе:

- ✓ Типы объектов стандартного веб-приложения
- ✓ Как различные типы объектов работают вместе
- ✓ На каком прикладном уровне находятся эти объекты
- ✓ Как связаны эти уровни

До сих пор мы обсуждали рекомендации по стилю для проектирования объектов. Предполагается, что это общие правила, которые могут применяться везде, но это не означает, что все объекты в приложении будут выглядеть одинаково. Одни объекты будут иметь множество методов-запросов, а другие — только командные методы. Некоторые будут сочетать те и другие в определенной пропорции. Можно заметить, что разные типы объектов часто имеют общие характеристики. Это приводит к тому, что для них создаются шаблонные названия. Например, разработчики будут говорить о «сущностях», «объектах-значениях» или «службах приложений», чтобы указать *природу* объекта, о котором идет речь.

В последней части этой книги рассматриваются распространенные типы объектов, которые встречаются в приложениях, и то, как их распознать в естественной среде. В этом смысле следующие разделы служат «справочником» объектов. Если

вы обнаружите объект, который не вписывается в определенную категорию, это руководство поможет решить, следует ли менять дизайн объекта, чтобы он лучше соответствовал остальным. Если вы столкнетесь с объектом, который не похож ни на один из тех, что описаны в этой главе, не волнуйтесь. До тех пор пока он соответствует рекомендациям по проектированию объектов из этой книги, все в порядке.

На рис. 10.1 представлен краткий обзор типов объектов, которые мы обсудим в следующих разделах. Если вы почувствуете, что запутались, обращайтесь к нему.

10.1. КОНТРОЛЛЕРЫ

В приложении всегда имеется *контроллер запросов*. Именно в него поступают все запросы. Если вы используете PHP, таким контроллером может служить index.php. В Spring framework в Java эту роль играет `DispatcherServlet` — на основе URI запроса, его метода, заголовков и т. д. вызов будет перенаправлен на *контроллер*, где приложение выполнит все, что ему нужно, прежде чем вернуть правильный ответ. Для приложений командной строки (CLI) «контроллером запросов» будет исполняемый файл, который вы вызываете, например bin/console, artisan и т. д. На основе аргументов, которые предоставляет пользователь, вызов будет перенаправлен на что-то наподобие *командного* объекта, где приложение может выполнить запрошенную задачу.

Хотя технически они совершенно разные, консольные команды концептуально очень похожи на веб-контроллеры. И те и другие выполняют работу, которая была запрошена извне пользователем или другим приложением, отправившим веб-запрос или запустившим консольное приложение. Так что будем называть контроллерами и консольные команды, и веб-контроллеры.

Контроллеры обычно содержат код, который показывает, откуда поступил вызов. Вам встретятся упоминания об объекте `Request`, параметрах запроса, формах, HTML-шаблонах, возможно, *сеансе* или файлах *cookie* (рис. 10.2). Все это веб-концепции. Классы, используемые здесь, часто происходят из веб-фреймворка приложения.

Другие контроллеры учитывают аргументы командной строки, параметры или флаги и содержат код для вывода строк текста в терминал и форматирования их способами, понятными терминалу (рис. 10.3). Все это указывает на то, что класс принимает входные данные и выдает выходные данные для командной строки.

Поскольку в случае контроллеров речь идет о конкретном механизме доставки, который инициировал вызов к ним (веб, терминал), контроллеры следует рассматривать как *инфраструктурный* код. Они облегчают связь между клиентом, который находится во *внешнем мире*, и *ядром* приложения.

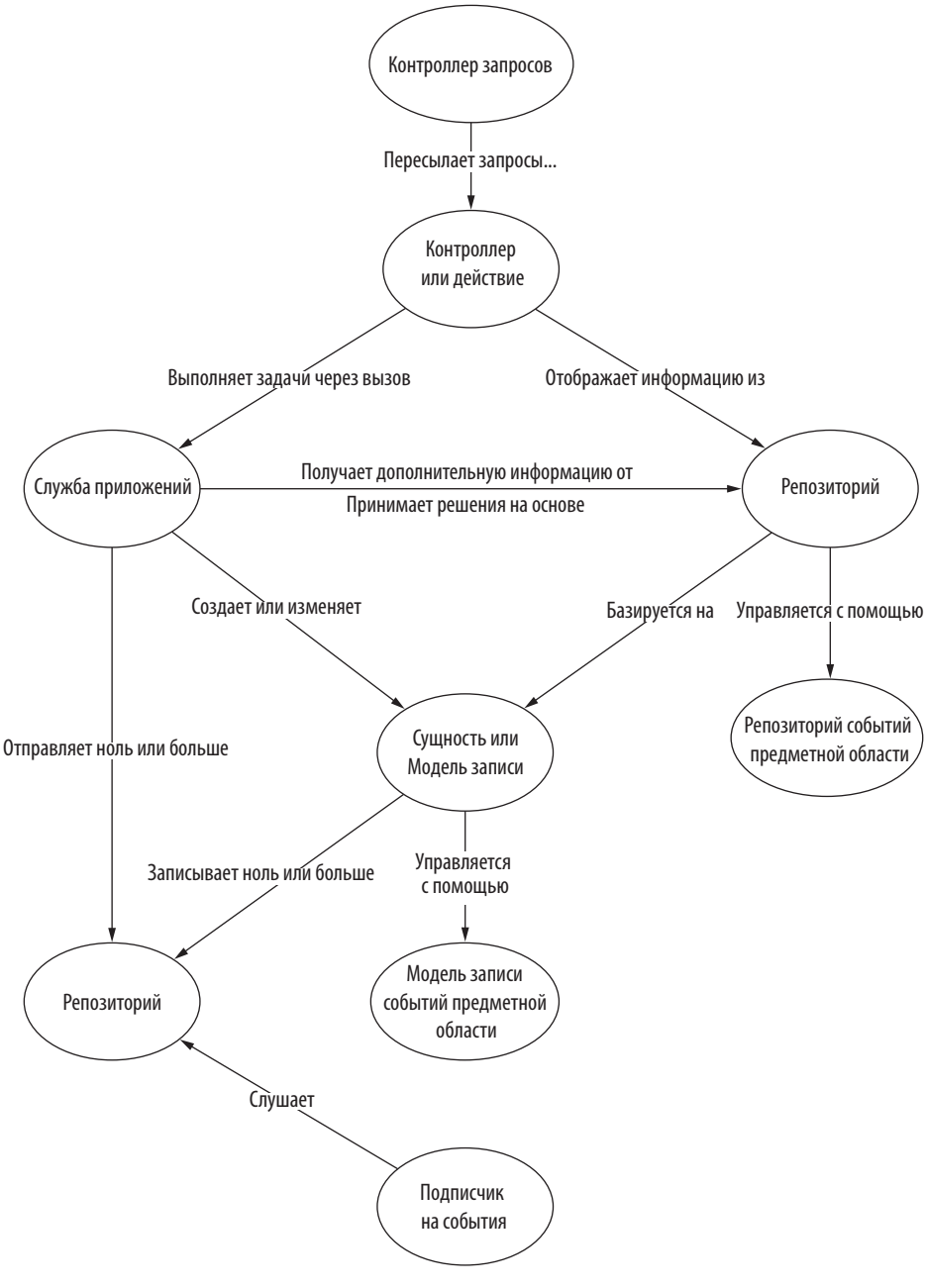


Рис. 10.1. Различные типы объектов и то, как они работают вместе в обычном (веб-) приложении

Рис. 10.2. Словесное облако терминов, встречающихся в веб-контроллере

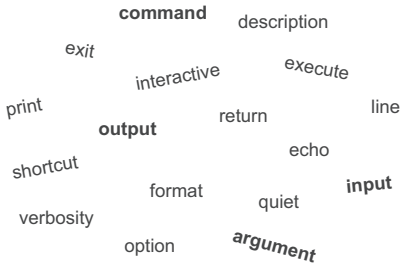


Рис. 10.3. Словесное облако терминов, встречающихся в консольной команде

Когда контроллер проверит предоставленные входные данные, он примет любую необходимую ему информацию, а затем вызовет либо *службу приложения*, либо *репозиторий моделей чтения*. Служба приложений будет вызвана, когда предполагается, что контроллер произведет некоторый эффект, например, когда предполагается внести изменения в состояние приложения, отправить электронное письмо и т. д. Репозиторий моделей чтения будет использоваться, если предполагается, что контроллер должен вернуть информацию, запрошенную клиентом.

ОБЪЕКТ ЯВЛЯЕТСЯ КОНТРОЛЛЕРОМ, ЕСЛИ...

- его вызывает внешний контроллер, и поэтому он является одной из точек входа для графа служб и их зависимостей (см. раздел 2.12);
- он содержит инфраструктурный код, который раскрывает механизм доставки;
- он вызывает службу приложения или репозиторий моделей чтения (или и то и другое).

Типичный веб-контроллер будет выглядеть примерно так, как показано в следующем листинге. Фреймворк, используемый в этих примерах, — Symfony (https://symfony.com/), надежный фреймворк для веб-приложений PHP.

Листинг 10.1. Типичный веб-контроллер

```
namespace Infrastructure\UserInterface\Web;

use Infrastructure\Web\Form\ScheduleMeetupType;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
```

```
use Symfony\Component\HttpFoundation\RedirectResponse;
use Symfony\Component\HttpFoundation\Response;
use Symfony\Component\HttpFoundation\Request;

final class MeetupController extends AbstractController
{
    public function scheduleMeetupAction(Request request): Response
    {
        form = this.createForm(ScheduleMeetupType.className);

        form.handleRequest(request);

        if (form.isSubmitted() && form.isValid()) {
            // ...

            return new RedirectResponse(
                '/meetup-details/' . meetup.meetupId()
            );
        }

        return this.render(
            'scheduleMeetup.html.twig',
             [
                'form' => form.createView()
            ]
        );
    }
}
```

Альтернативный контроллер для командной строки может выглядеть, как показано ниже.

Листинг 10.2. Типичный контроллер командной строки, или консольная команда

```
namespace Infrastructure\UserInterface\Cli;

use Symfony\Component\Console\Command\Command;
use Symfony\Component\Console\Input\InputInterface;
use Symfony\Component\Console\Output\OutputInterface;

final class ScheduleMeetupCommand extends Command
{
    protected function configure()
    {
        this
            .addArgument('title', InputArgument.REQUIRED)
            .addArgument('date', InputArgument.REQUIRED)
            // ...
        ;
    }
```

```
    public function execute(
        InputInterface input,
        OutputInterface output
    ) {
        title = input.getArgument('title');
        date = input.getArgument('date');

        // ...

        output.writeln('Meetup scheduled');
    }
}
```

10.2. СЛУЖБЫ ПРИЛОЖЕНИЙ

Служба приложений представляет собой задачу, которую необходимо выполнить. Любая зависимость вводится в качестве аргумента конструктора. Все соответствующие данные, необходимые для выполнения задачи (см. раздел 2.8), включая контекстную информацию (такую, как идентификатор пользователя, вошедшего в систему, или текущее время), предоставляются в качестве аргументов метода. Когда данные поступают от самого клиента, это данные примитивного типа. Таким образом, контроллер может предоставлять службе приложений данные в том виде, в каком они были отправлены клиентом, без их предварительного преобразования.

Код службы приложений должен читаться как пошаговый рецепт со всеми действиями, необходимыми для выполнения задачи. Например: «Извлеките объект из такого-то репозитория моделей записи, вызовите для него метод и сохраните его снова». Или: «Соберите такую-то информацию из этого репозитория моделей чтения и отправьте отчет определенному пользователю».

ОБЪЕКТ ЯВЛЯЕТСЯ СЛУЖБОЙ ПРИЛОЖЕНИЯ, ЕСЛИ...

- он выполняет единственную задачу;
- он не содержит инфраструктурного кода, то есть самостоятельно не работает с веб-запросом, или SQL-запросом, или файловой системой и т. д.;
- в нем описывается единственный вариант использования, который должно иметь приложение. Часто это точно соответствует запросу функции от заинтересованной стороны. Например, должна присутствовать возможность добавить товар в каталог, отменить заказ, отправить покупателю накладную и т. д.

Веб-контроллер и обработчик консоли, которые мы видели в листингах 10.1 и 10.2, будут принимать данные из запроса (через форму) или из аргументов командной строки и передавать их в службу приложений, которая выглядит примерно так.

Листинг 10.3. Служба приложений

```
namespace Application\ScheduleMeetup;

use Domain\Model\Meetup\Meetup;
use Domain\Model\Meetup\MeetupRepository;
use Domain\Model\Meetup\ScheduleDate;
use Domain\Model\Meetup\Title;

final class ScheduleMeetupService
{
    private MeetupRepository meetupRepository;

    public function __construct(MeetupRepository meetupRepository)
    {
        this.meetupRepository = meetupRepository;
    }

    public function schedule(
        string title,
        string date,
        UserId currentUserId
    ): MeetupId {
        meetup = Meetup.schedule(
            this.meetupRepository.nextIdentity(),
            Title.fromString(title),
            ScheduledDate.fromString(date),
            currentUserId
        );

        this.meetupRepository.save(meetup);

        return meetup.meetupId();
    }
}
```

Служба приложений получает аргументы примитивного типа

Она преобразует эти значения примитивного типа в объекты-значения и создает новую сущность Meetup с использованием этих объектов

Митап сохраняется в репозиторий моделей записи

Наконец, она возвращает идентификатор нового объекта Meetup

Иногда службы приложений называются обработчиками команд, но они все равно остаются службами приложений. Вместо вызова службы с использованием аргументов примитивного типа ее можно вызвать, предоставив *командный объект*, содержащий запрос клиента в едином объекте. Такой объект называется *объектом для передачи данных* (DTO), поскольку используется для передачи данных, предоставленных клиентом, в виде одного объекта от контроллера к службе приложения. Это должен быть простой в построении объект, который содержит только значения примитивного типа, простые списки и, возможно, другие DTO, если требуется иерархия.

Листинг 10.4. Пример передачи DTO при вызове службы приложения

```
namespace Application\ScheduleMeetup;

final class ScheduleMeetup
{
    public string title;
    public string date;
}

final class ScheduleMeetupService
{
    // ...

    public function schedule(
        ScheduleMeetup command,
        UserId currentUserId
    ): MeetupId {
        meetup = Meetup.schedule(
            this.meetupRepository.nextIdentity(),
            Title.fromString(command.title),
            ScheduledDate.fromString(command.date),
            currentUserId
        );

        // ...
    }
}
```

Эта команда содержит данные, необходимые для выполнения задачи планирования митапа

Служба приложения может затем взять данные из командного объекта

Преимущество использования выделенного командного объекта заключается в том, что его легко инстанцировать на основе десериализованных строковых данных, таких как тело запроса JSON или XML. Он также хорошо работает с библиотеками форм, которые могут отображать отправленные данные непосредственно в свойства DTO команды.

10.3. РЕПОЗИТОРИИ МОДЕЛЕЙ ЗАПИСИ

Часто служба приложения вносит изменения в состояние приложения, и это обычно означает, что объект предметной области необходимо изменить и сохранить. Служба приложения использует для этого абстракцию: *репозиторий*. Если точнее, то это *репозиторий моделей записи*, поскольку он связан только с извлечением объекта и сохранением внесенных в него изменений.

Сама абстракция будет представлять собой интерфейс, который служба вводит как зависимость. Этот интерфейс не предоставляет подробностей о том, *как именно* объект будет сохранен. Он просто предлагает ряд методов общего назначения, таких как `GetById()`, `save()`, `add()` или `update()`. Соответствующая реализация

возьмет на себя детали, например, какие SQL-запросы будут выдаваться или какой ORM будет использоваться для сопоставления объекта со строкой в базе данных.

ОБЪЕКТ ЯВЛЯЕТСЯ РЕПОЗИТОРИЕМ МОДЕЛЕЙ ЗАПИСИ, ЕСЛИ...

- он предлагает методы для извлечения объекта из хранилища и для его сохранения;
- его интерфейс скрывает использованную базовую технологию.

В качестве примера в следующем листинге показан репозиторий `MeetupRepository`, на который опирается служба приложения в листинге 10.3.

Листинг 10.5. Интерфейс репозитория моделей записи и его реализация

```
namespace Domain\Model\Meetup;

interface MeetupRepository
{
    public function save(Meetup meetup): void;

    public function nextIdentity(): MeetupId;

    /**
     * @throws MeetupNotFound
     */
    public function getById(MeetupId meetupId): Meetup
}

namespace Infrastructure\Persistence\DoctrineOrm;

use Doctrine\ORM\EntityManager;
use Domain\Model\Meetup\Meetup;
use Domain\Model\Meetup\MeetupId;
use Ramsey\Uuid\UuidFactoryInterface;

final class DoctrineOrmMeetupRepository
    implements MeetupRepository
{
    private EntityManager entityManager;
    private UuidFactoryInterface uuidFactory;

    public function __construct(
        EntityManager entityManager,
        UuidFactoryInterface uuidFactory
    ) {
        this.entityManager = entityManager;
        this.uuidFactory = uuidFactory;
    }
```

Реализация MeetupRepository по умолчанию использует Doctrine ORM

```
    public function save(Meetup meetup): void
    {
        this.entityManager.persist(meetup);
        this.entityManager.flush(meetup);
    }

    public function nextIdentity(): MeetupId
    {
        return MeetupId.fromString(
            this.uuidFactory.uuid4().toString()
        );
    }

    // ...
}
```

10.4. СУЩНОСТИ

Сохраняемые объекты — это нужные пользователю объекты, о которых следует помнить, даже когда приложение необходимо перезапустить. Это *сущности* приложения.

Сущности представляют собой понятия из предметной области приложения. Они включают в себя соответствующие данные и полезное поведение, связанное с этими данными. С точки зрения проектирования объектов, они часто будут содержать именованные конструкторы, поскольку это позволяет использовать имена из предметной области для создания объектов конкретного типа (см. раздел 3.9). У них также будут методы-модификаторы, которые представляют собой командные методы, изменяющие состояние сущности (см. раздел 4.6). Сущности содержат лишь несколько методов-запросов, если они вообще есть. Извлечение информации обычно привязывается к определенному виду объекта, называемому объектом запроса. Мы еще вернемся к этому.

ПРАВИЛЬНЫЕ СУЩНОСТИ

Как и любой объект, сущность изо всех сил защищает себя от попадания в недопустимое состояние. Согласно этому определению, многие физические сущности не следует считать правильными.

Когда изменение состояния разрешено, сущность обычно создает событие предметной области, представляющее это изменение (см. раздел 4.12). Эти события можно использовать, чтобы выяснить, что именно изменилось, и сообщить об этом изменении другим частям приложения, которым следует на это отреагировать.

ОБЪЕКТ ЯВЛЯЕТСЯ СУЩНОСТЬЮ, ЕСЛИ ОН...

- имеет уникальный идентификатор;
- имеет жизненный цикл;
- сохраняется в репозитории моделей записи и позже может быть извлечен из него;
- использует именованные конструкторы и методы команд, чтобы предоставить пользователю способы инстанцирования его экземпляра и управления его состоянием;
- создает события предметной области во время своего инстанцирования или изменения.

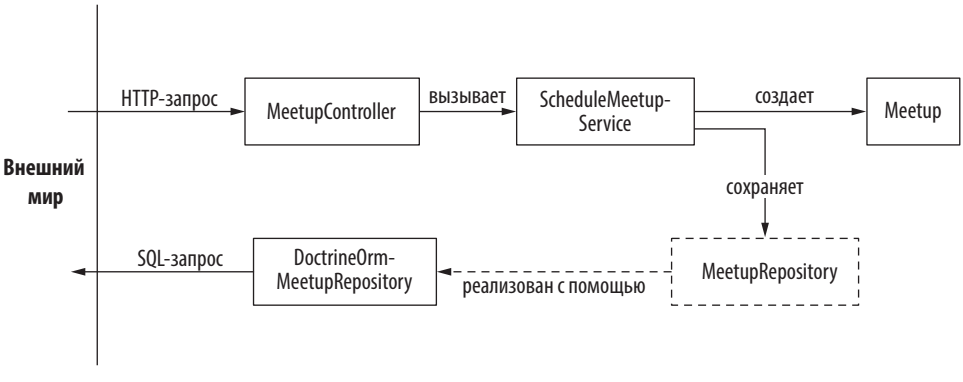


Рис. 10.4. Все рассмотренные объекты работают вместе, чтобы запланировать митап

10.5. ОБЪЕКТЫ-ЗНАЧЕНИЯ

Объекты-значения являются оболочками для значений примитивного типа, придавая этим значениям смысл и полезное поведение. Мы подробно обсуждали их ранее (глава 3). В контексте перехода от контроллера к службе приложения и репозиторию следует отметить, что часто именно служба приложения инстанцирует экземпляры объектов-значений, а затем передает их в качестве аргументов конструктору или методу-модификатору сущности. Следовательно, они в конечном итоге используются или хранятся внутри объекта.

Однако полезно помнить, что объекты-значения предназначены не только для использования в сочетании с сущностями. Их можно применять в любом месте,

и объект-значение на самом деле является предпочтительным способом передачи значений.

ОБЪЕКТ — ЭТО ОБЪЕКТ-ЗНАЧЕНИЕ, ЕСЛИ ОН...

- неизменяемый;
- обертывает данные примитивного типа;
- добавляет смысл, используя термины, относящиеся к конкретной предметной области (например, не просто `int`, а `Year`);
- накладывает ограничения посредством проверки (например, не просто строка, а строка с символом "@");
- действует как аттрактор полезного поведения, связанного с концепцией (например, `Position.toTheLeft(int steps)`).

Объект `Meetup`, который, как мы видели, был создан в листинге 10.3 вместе со связанными с ним объектами-значениями и событиями предметной области, выглядит примерно так.

Листинг 10.6. Сущность

```
namespace Domain\Model\Meetup;

final class Meetup
{
    private array events = [];

    private MeetupId meetupId;
    private Title title;
    private ScheduledDate scheduledDate;
    private UserId userId;

    private function __construct()
    {
    }

    public static function schedule(
        MeetupId meetupId,
        Title title,
        ScheduledDate scheduledDate,
        UserId userId
    ): Meetup {
        meetup = new Meetup();

        meetup.meetupId = meetupId;
        meetup.title = title;
```

```
        meetup.scheduledDate = scheduledDate;
        meetup.userId = userId;

        meetup.recordThat(
            new MeetupScheduled(
                meetupId,
                title,
                scheduledDate,
                userId
            );
        );

        return meetup;
    }

    public function reschedule(ScheduledDate scheduledDate): void
    {
        // ...

        this.recordThat(
            new MeetupRescheduled(this.meetupId, scheduledDate)
        );
    }

    public function cancel(): void
    {
        // ...
    }

    // ...

    private function recordThat(object event): void
    {
        this.events[] = event;
    }

    public function releaseEvents(): array
    {
        return this.events;
    }

    public function clearEvents(): void
    {
        this.events = [];
    }
}

final class Title
{
    private string title;

    private function __construct(string title)
    {
```

Следующие методы являются примерами другого поведения, которое мог бы предложить объект Meetup

```
        Assertion.notEmpty(title);
        this.title = title;
    }

    public static function fromString(string title): Title
    {
        return new Title(title);
    }

    public function abbreviated(string ellipsis = '...'): string
    {
        // ...
    }
}

final class MeetupId
{
    private string meetupId;

    private function __construct(string meetupId)
    {
        Assertion.uuid(meetupId);
        this.meetupId = meetupId;
    }

    public static function fromString(string meetupId): MeetupId
    {
        return new MeetupId(meetupId);
    }
}
```

С таким же успехом можно использовать обычный общедоступный конструктор…

Это пример полезного поведения, которое объект-значение помогает добавлять

10.6. ПРОСЛУШИВАТЕЛИ СОБЫТИЙ

Мы уже говорили о событиях предметной области. Их можно использовать для уведомления других сервисов о событиях, произошедших внутри модели записи. После выполнения основной задачи эти другие сервисы могут выполнять вторичные действия. Поскольку основные задачи выполняются службами приложений, события предметной области можно использовать для уведомления других сервисов уже *после* завершения работы службы приложения. Они также могут делать это в последний момент, непосредственно перед возвращением. На этом этапе служба приложения может извлекать записанные события из объекта, который она изменила, и передавать их *диспетчеру событий*, как показано в следующем листинге.

Листинг 10.7. Служба приложения отправляет события предметной области

```
final class RescheduleMeetupService
{
    private EventDispatcher dispatcher;
```

```
    public function __construct(
        // ...
        EventDispatcher dispatcher
    ) {
        this.dispatcher = dispatcher
    }

    public function reschedule(MeetupId meetupId, /* ... */): void
    {
        meetup = /* ... */;

        meetup.reschedule(/* ... */);

        this.dispatcher.dispatchAll(meetup.recordedEvents());
    }
}
```

Отправка любого события, которое записано внутри объекта Meetup

Внутри диспетчер будет пересылать все события сервисам, называемым *прослушивателями* или *подписчиками*, которые были зарегистрированы для определенных типов событий.

Затем прослушиватель событий выполняет вторичные действия, для которых он может даже вызвать другую службу приложения. Он может использовать любой необходимый ему сервис, например отправлять электронные письма с уведомлением о только что произошедшем событии. Возьмем, к примеру, прослушиватель `NotifyGroupMembers`, который будет уведомлять участников группы о переносе митапа.

Листинг 10.8. Прослушиватель событий отвечает на события предметной области

```
final class NotifyGroupMembers {
    public function whenMeetupRescheduled(
        MeetupRescheduled event
    ): void {
        /*
         * Отправить электронное письмо участникам группы, используя информацию
         * из объекта события.
         */
    }
}
```

Удобный способ именования прослушивателей событий — это описание того, что вы собираетесь сделать (например, «уведомить участников группы»). Затем в методах указываются причины (например, «когда митап перенесен»)

ОБЪЕКТ — ЭТО ПРОСЛУШИВАТЕЛЬ СОБЫТИЙ, ЕСЛИ...

- он представляет собой неизменяемый сервис с внедренными зависимостями;
- у него есть хотя бы один метод, который принимает единственный аргумент, являющийся событием предметной области.

10.7. МОДЕЛИ ЧТЕНИЯ И РЕПОЗИТОРИИ МОДЕЛЕЙ ЧТЕНИЯ

Как мы уже говорили, контроллер может вызывать службу приложения для выполнения задачи, а также *репозиторий моделей чтения* для извлечения информации. Такой репозиторий будет возвращать объекты. Эти объекты предназначены не для манипулирования, а для получения из них информации. Ранее мы называли их объектами запроса. В них содержатся только методы-запросы, это означает, что пользователи не могут влиять на их состояние.

Когда вызов репозитория моделей чтения происходит внутри контроллера, возвращаемая модель чтения может быть передана средству визуализации шаблонов, которое может сгенерировать HTML-ответ с его использованием. Или его можно легко использовать для генерации ответа в кодировке JSON на вызов API. Во всех этих случаях модель чтения специально разработана для соответствия ответу, который будет сгенерирован. Все данные, необходимые для конкретного варианта использования, должны быть доступны внутри модели чтения, и никаких дополнительных запросов делать не нужно. Такая модель чтения является DTO, потому что она будет использоваться для *передачи* данных из ядра приложения во внешний мир. Значения, которые могут быть извлечены из такой модели чтения, должны иметь примитивные типы.

В качестве примера возьмем следующий репозиторий моделей чтения, который возвращает список предстоящих митапов. Он предназначен для конкретного варианта использования и содержит только данные, необходимые для отображения простого списка.

Листинг 10.9. Модель чтения и ее репозиторий

```
namespace Application\UpcomingMeetups;

final class UpcomingMeetup
    public string title;
    public string date;
}

interface UpcomingMeetupRepository
{
    /**
     * @return UpcomingMeetup[]
     */
    public function upcomingMeetups(DateTime today): array
}

namespace Infrastructure\ReadModel;
```

UpcomingMeetup — это модель чтения (или «модель просмотра») — DTO, содержащий соответствующие данные о предстоящих митапах, которые будут отображаться в списке на веб-странице

Она поставляется с репозиторием, который возвращает экземпляры UpcomingMeetup и может использоваться веб-контроллером и передаваться в средство визуализации шаблонов

```
use Application\UpcomingMeetups\UpcomingMeetupRepository;
use Doctrine\DBAL\Connection;

final class UpcomingMeetupDoctrineDbalRepository implements
    UpcomingMeetupRepository
{

    private Connection connection;

    public function __construct(Connection connection)
    {
        this.connection = connection;
    }

    public function upcomingMeetups(DateTime today): array
    {
        rows = this.connection./* ... */;

        return array_map(
            function (array row) {
                upcomingMeetup = new UpcomingMeetup();
                upcomingMeetup.title = row['title'];
                upcomingMeetup.date = row['date'];

                return upcomingMeetup;
            },
            rows
        );
    }
}
```

Эта реализация UpcomingMeetupRepository извлекает данные непосредственно из базы данных. Затем она создает экземпляры модели чтения UpcomingMeetup

Сама служба приложения также может задействовать репозиторий моделей чтения для извлечения информации. Затем она может использовать эту информацию для принятия решений или выполнения дальнейших действий. Модель чтения, используемая службой приложения, часто является более «интеллектуальной», нежели та, которая применяется для генерации ответа. Для возвращаемых значений она использует соответствующие объекты-значения вместо значений примитивного типа, поэтому службе приложения не нужно беспокоиться о ее достоверности. Может показаться, что такая модель чтения является моделью записи, за исключением того что в нее невозможно внести изменения; в конце концов, это объект запроса.

Что касается репозиториев моделей чтения, то их необходимо разделить на абстракцию и конкретную реализацию. Как и в случае с репозиториями моделей записи, интерфейс предложит один или несколько методов-запросов, которые можно использовать для извлечения моделей чтения. Интерфейс никак не раскрывает механизма хранения этих моделей.

ОБЪЕКТ — ЭТО РЕПОЗИТОРИЙ МОДЕЛЕЙ ЧТЕНИЯ, ЕСЛИ...

- у него есть методы-запросы, которые соответствуют конкретному варианту использования и будут возвращать модели чтения, которые также специфичны для этого варианта использования.

ОБЪЕКТ — ЭТО МОДЕЛЬ ЧТЕНИЯ, ЕСЛИ...

- у него есть только методы-запросы, т. е. это объект запроса (и, следовательно, он неизменяемый);
- он разработан специально для определенного варианта использования;
- все необходимые данные (и никакие другие) становятся доступными при извлечении объекта.

Обратите внимание, что различие между репозиторием моделей чтения и обычным сервисом, который возвращает фрагмент информации, не так очевидно. Например, рассмотрим ситуацию, когда службе приложения требуется обменный курс, чтобы конвертировать определенную сумму денег в иностранную валюту. Можно сказать, что сервис, который умеет предоставлять такую информацию, обычно является репозиторием, из которого можно получить обменный курс для данной валюты. Такой сервис имеет доступ к «коллекции» обменных курсов, заданной в месте, которое для нас не имеет значения. Тем не менее этот сервис также можно считать обычным и с таким же успехом называть его `ExchangeRateProvider` или похожим образом.

Основная идея заключается в том, что для всех этих сервисов требуются абстракция (пример см. в следующем листинге) и конкретная реализация, потому что абстракция описывает, что необходимо найти, а реализация — как это получить.

Листинг 10.10. Обычная служба

```
namespace Application\ExchangeRates;

interface ExchangeRateProvider
{
    public function getRateFor(
        Currency from,
        Currency to
    ): ExchangeRate;
}

final ExchangeRate
{
    // ...
}
```

Абстракция — это интерфейс, представляющий собой задаваемый вопрос

Типы возвращаемых значений, используемых интерфейсом, также являются частью абстракции, потому что нас интересует то, как использовать эти значения, но не то, как в них попадают данные

С точки зрения дизайна некоторые объекты не сильно отличаются друг от друга. Например, события предметной области очень похожи на объекты-значения — это неизменяемые объекты, содержащие привязанные к ним данные. Разница между событием и объектом-значением заключается в том, как и где они используются: событие предметной области будет создано и записано внутри сущности, а затем отправлено; объект-значение моделирует аспект сущности.

10.8. АБСТРАКЦИИ, КОНКРЕТИКА, СЛОИ И ЗАВИСИМОСТИ

До сих пор мы рассматривали типы объектов, которые можно встретить в обычном веб- или консольном приложении. Помимо определенных характеристик (например, типы методов, которыми обладают эти объекты, какую информацию они предоставляют или какое поведение предлагают), необходимо учитывать, являются ли они *абстрактными* или *конкретными* и каким образом эти объекты *зависят* друг от друга.

С точки зрения абстракции можно определить следующие характеристики типов объектов, которые мы обсуждали до сих пор:

- *Контроллеры конкретны.* Они часто связаны с определенным фреймворком и специфичны для механизма доставки. У них нет интерфейса, и он им не нужен. Единственный вариант, когда вам понадобится альтернативная реализация, — это переключение на другой фреймворк. В этом случае перепишите контроллеры вместо того, чтобы создавать для них вторую реализацию.
- *Службы приложения конкретны.* Они представляют собой очень специфический вариант использования приложения. Если вариант использования меняется, меняется и сама служба, поэтому у них нет интерфейса.
- *Сущности и объекты-значения конкретны.* Они являются конкретным результатом понимания разработчиком предметной области. Эти типы объектов *развиваются* с течением времени. Мы не предоставляем для них интерфейс. То же самое относится и к объектам модели чтения. Мы определяем и используем их такими, какие они есть, а не через интерфейс.
- *Репозитории (для моделей записи и чтения) состоят из абстракции и, по крайней мере, одной конкретной реализации.* Репозитории — это службы, которые подключаются к чему-то за пределами приложения, например к базе данных, файловой системе или удаленному сервису. Вот почему им нужна абстракция, которая представляет, что будет делать сервис и что он вернет. Затем реализация предоставляет все низкоуровневые сведения о том, как именно он должен это делать. То же самое относится и к другим объектам, которые будут подключаться к сервису вне приложения. Этим сервисам также потребуются интерфейс и конкретная реализация.

Сервисы, для которых имеются абстракции, согласно списку выше, должны *вводиться* как абстрактные зависимости. Поступая так, можно сформировать три полезные группы, или *слоя*, объектов:

1. *Инфраструктурный* слой:
   - Контроллеры.
   - *Реализации* моделей записи и чтения.
2. *Прикладной* слой:
   - Службы приложения.
   - Командные объекты.
   - Модели чтения.
   - *Интерфейсы* репозиториев моделей чтения.
   - Прослушиватели событий.
3. Слой *предметной области*:
   - Сущности.
   - Объекты-значения.
   - *Интерфейсы* репозиториев моделей записи.

Учитывая, что *инфраструктурный* слой содержит код, облегчающий связь с *внешним миром*, его можно изобразить как слой вокруг *приложения* и *предметной области* (рис. 10.5). Аналогично *приложение* использует код на уровне слоя *предметной области* для выполнения своих задач, поэтому слой предметной области будет самым внутренним слоем приложения.

Рис. 10.5. Слои можно визуализировать в виде концентрических кругов

Чтобы продемонстрировать использование слоев в коде, можно сделать имена слоев частью пространств имен классов. Примеры кода в этой главе также используют это соглашение. Вводя абстрактные зависимости, можно гарантировать, что объекты будут зависеть друг от друга только в одном направлении: сверху вниз. Например, служба приложения, которой требуется репозиторий моделей записи, будет зависеть от интерфейса этого репозитория, а не от его конкретной реализации. Это дает два основных преимущества.

Во-первых, можно протестировать код службы приложения без реальной реализации репозитория, для чего потребовалась бы запущенная база данных с правильной схемой и т. д. У нас есть интерфейсы для всех этих служб, и создавать для них тестовые дубликаты легко.

Во-вторых, легко переключать реализации инфраструктуры. Слой приложения выдержит переключение между фреймворками (или обновление до следующей основной версии фреймворка), а также переключение баз данных (когда вы поймете, что лучше использовать графовую базу данных вместо, например, реляционной) и удаление служб (когда вы будете получать обменные курсы не из внешней службы, а из собственной локальной базы данных).

ЗАКЛЮЧЕНИЕ

- Контроллер запросов приложения перенаправляет входящий запрос на один из своих контроллеров. Эти контроллеры являются частью *инфраструктурного слоя* приложения и знают, как преобразовать входящие данные в вызов службы приложения или репозитория моделей чтения, которые являются частью *прикладного слоя*.
- Служба приложения не зависит от механизма доставки, и ее так же легко использовать в веб-приложениях или консольных приложениях. Она выполняет единственную задачу, которая представляет собой один из вариантов использования приложения. Попутно она может извлечь сущность из репозитория моделей записи, вызвать для нее метод и сохранить ее измененное состояние. Сама сущность, включая ее объекты-значения, является частью *слоя предметной области*.
- Репозиторий моделей чтения — это сервис, который можно использовать для извлечения информации. Он возвращает модели чтения, специфичные для конкретного варианта использования и предоставляющие всю необходимую информацию (и ничего больше).
- Типы объектов, описанные в этой главе, естественным образом принадлежат к тому или иному слою. Многослойная система, в которой код зависит только от кода на более низких уровнях, служит способом отделить код предметной области и приложения от инфраструктурных аспектов приложения.

11 Эпилог

В этой главе:

- ✓ Что еще почитать об архитектурных шаблонах
- ✓ Улучшение стратегии тестирования
- ✓ Советы и информация о предметно-ориентированном проектировании

Цель этой книги — стать руководством по стилю. Она содержит основные правила проектирования объектов, отражаемые в объявлениях классов и методов. Для многих таких правил можно создать инструмент статического анализа, выдающий предупреждения, когда они не соблюдаются. Этот инструмент мог бы, например, предупреждать о методах, которые вносят изменения в состояние и что-то возвращают. Или о сервисах с методами, которые меняют свое поведение после создания.

Здесь следует сделать две оговорки. Во-первых, я считаю, что следовать правилам важно, но в некоторых особых случаях можно позволить себе нарушить их, например, когда качество на самом деле не имеет значения, потому что не придется долго поддерживать код. Или когда соблюдение *всех* правил потребует очень многих усилий, и возможная выгода не оправдает этих затрат. Однако не спешите меня осуждать. Я считаю, что в 95 % реальных сценариев легких путей не существует.

Во-вторых, правила — еще не все, что нужно для проектирования объектов. Они не скажут точно, какие объекты вам понадобятся, какими должны быть их функции и т. д. Для меня правила, изложенные в этой книге, — это правила, по которым я живу, почти не задумываясь. Благодаря им появляется больше возможностей пробовать что-то новое, направлять умственную энергию в другие области.

В этой главе я хотел бы отметить еще одну тему, которая помогает частично облегчить когнитивную нагрузку при разработке приложений: архитектурные шаблоны. Я также назову две темы, в которые стоит углубиться после прочтения книги: тестирование и предметно-ориентированное проектирование (domain-driven design, DDD). Обе эти сферы помогут узнать больше о дизайне объектов.

11.1. АРХИТЕКТУРНЫЕ ШАБЛОНЫ

В предыдущей главе мы обсуждали, как определенные типы объектов образуют естественный набор слоев. Помимо использования слоев для структурирования приложения в целом (что следует рассматривать как вопрос архитектуры), важно знать о том, как приложение связано с внешним миром. Определение способов взаимодействия для приложения приводит к четкому разделению между кодом, который поддерживает это взаимодействие, и кодом, который служит ядром приложения. Такой подход к архитектуре называется *гексагональной архитектурой*, или, иногда, методом *портов и адаптеров*.

По этой теме я рекомендую главу 4 книги Вона Вернона (Vaughn Vernon) «Implementing Domain-Driven Design»[1] (Addison-Wesley Professional, 2013). Несколько моих статей также посвящены гексагональной архитектуре:

- “Layers, ports & adapters — Part 2, Layers”, http://mng.bz/2Jao;
- “Layers, ports & adapters — Part 3, Ports & Adapters”, http://mng.bz/1wMQ;
- “When to add an interface to a class”, http://mng.bz/POx8.

11.2. ТЕСТИРОВАНИЕ

В этой книге мы обсуждали проектирование объектов, а также рассмотрели несколько методов тестирования. Проектировать объекты и одновременно тестировать их очень удобно. Когда вы примените подход «сначала тест», то обнаружите, что пишете только тот код, который вам действительно нужен для реализации желаемого поведения. Тесты доказывают, что созданные объекты можно использовать так, как вы себе это представляли. И всякий раз, когда вы

[1] Вернон В. «Реализация методов предметно-ориентированного проектирования».

думаете о возможных исключениях или сталкиваетесь с ошибками в коде, лежащем в основе объектов, вы можете описать ситуацию в тестовом примере, увидеть сбой, а затем исправить код.

11.2.1. Тестирование класса в сравнении с тестированием объекта

Обратите внимание, что я говорю о тестировании *объектов*. Я считаю, что разработчики, в том числе я сам, часто склонны тестировать *классы*, а не объекты. Это различие может показаться незначительным, но оно влечет ряд довольно серьезных последствий. При тестировании *классов* обычно тестируется один метод одного класса со всеми его зависимостями, подменяемыми тестовыми дублерами. Такой тест в конечном итоге оказывается слишком близким к реализации. Он проверяет, выполняются ли вызовы методов, добавляет геттеры для получения данных из объекта и т. д.

Тесты, которые тестируют *классы*, можно рассматривать как тесты *белого ящика*, в отличие от тестов *черного ящика*, которые определенно предпочтительны. Тест черного ящика проверяет поведение объекта, как оно выглядит извне, без знаний внутренних особенностей класса. Он инстанцирует экземпляр объекта только с тестовыми дублерами объектов, пересекающих границу системы. В остальном все как в жизни. Такие тесты показывают, что не только отдельный класс, но и более крупная единица кода хорошо работает в целом.

Тесты классов постоянно меняются согласно изменениям, вносимым в сами классы. Объектные тесты в большей степени отделены от реализации тестируемого объекта, поэтому они более полезны в долгосрочной перспективе. Собственно, вот правило тестирования, которому стоит следовать: пишите тесты так, чтобы можно было изменить как можно больше деталей реализации, прежде чем менять код самого теста.

11.2.2. Разработка функций сверху вниз

Еще одна вещь, о которой следует помнить при тестировании ПО, — это уровень детализации. Я замечаю, что разработчики, как и я сам, часто предпочитают работать с более мелкими деталями — строительными блоками, которые затем можно использовать для завершения функции. Мы часто думаем обо всем, что нам понадобится для полноценной работы, и собираем все ингредиенты воедино: создаем репозиторий, таблицу базы данных, объект и т. д. Как только мы пытаемся объединить все составляющие, то обнаруживаем, что теперь их нужно пересматривать, потому что из-за нескольких неправильных предположений они плохо работают вместе. Можно сказать, что эти усилия по разработке потрачены впустую.

Я рекомендую действовать наоборот и начинать с более широкой картины. Определите, как будет использоваться функция: опишите пользовательские сценарии, сделайте наброски взаимодействия и т. д. Другими словами, задайте высокоуровневое поведение приложения таким, каким вы ожидаете его видеть после завершения всей работы. Не погружайтесь сходу в низкоуровневые детали реализации. Как только вы поймете, на что должно быть способно приложение, представленное как черный ящик, вы сможете обратиться к более глубоким слоям и написать код для всего, что необходимо.

Определять поведение приложения необходимо с помощью тестов, так что нисходящий стиль разработки полностью управляется тестированием. Высокоуровневые тесты, описывающие завершенную функцию, не будут пройдены до тех пор, пока не закончатся все тесты более низкого уровня. На рис. 11.1 показано, как можно завершить функцию путем прохождения тестов более низкого уровня, в то же время постепенно проходя тесты высокого уровня.

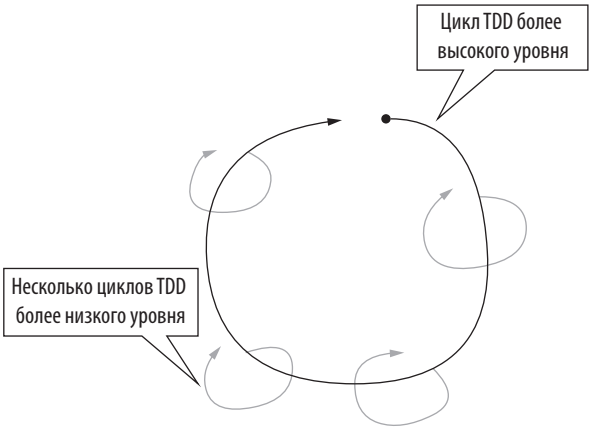


Рис. 11.1. Цикл TDD более высокого уровня подходит к концу после успешного завершения нескольких циклов TDD более низкого уровня

Отличная книга, в которой разбирается этот подход, — «Growing Object-Oriented Software, Guided by Tests» Стива Фримена и Ната Прайса (Steve Freeman, Nat Pryce) (Addison-Wesley Professional, 2009).

Если вы согласуете нисходящий подход к разработке со своим подходом к тестированию, то сможете определить автоматизируемые критерии готовности продукта. Они покажут, что вы создали действительно все, что необходимо.

Чтобы узнать больше об этой увлекательной теме, обратитесь к изданиям «Specification by Example: How Successful Teams Deliver the Right Software» (Manning, 2011) и «Bridging the Communication Gap: Specification by Example and Agile

Acceptance Testing» (Neuri, 2009), оба за авторством Гойко Аджича (Gojko Adzic); и «Discovery: Explore Behaviour Using Examples» (BDD Books, 2018) Гашпара Надя и Себа Роза (Gáspár Nagy, Seb Rose) (которое является частью продолжающейся серии).

11.3. ПРЕДМЕТНО-ОРИЕНТИРОВАННОЕ ПРОЕКТИРОВАНИЕ

Если вы ищете больше подсказок о том, какие типы объектов нужно добавить в приложение, то вам непременно стоит обратить внимание на предметно-ориентированное проектирование (DDD). В его основе лежит идея о том, чтобы получать знания о предметной области, а затем отражать их в модели предметной области приложения. *Предметно-ориентированный подход* помогает сосредоточиться на дизайне, переключая внимание от инфраструктурных деталей, таких как таблицы и столбцы базы данных.

Хотя стратегический аспект DDD тоже довольно увлекателен, вы увидите, что самые полезные предложения с точки зрения дизайна объектов содержатся в тактических рекомендациях. Обратитесь к книгам «Domain-Driven Design: Tackling Complexity in the Heart of Software» Эрика Эванса (Eric Evans)[1] (Addison-Wesley Professional, 2003) и «Implementing Domain-Driven Design» Вона Вернона (Addison-Wesley Professional, 2013). Они содержат множество практических советов по проектированию сущностей и объектов-значений, а также других связанных типов объектов.

11.4. ЗАКЛЮЧЕНИЕ

Конечно, в проектировании объектов, а также разработке ПО и архитектуре в целом есть еще множество интересного. Если говорить о проектировании объектов, я надеюсь, что эта книга дала вам прочную основу и несколько полезных советов на пути к дальнейшему изучению этой темы. Вы обнаружите, что с каждым днем узнаёте все больше и больше, так что продолжайте экспериментировать! Желаю вам удачи!

---

[1] Эванс Э. «Предметно-ориентированное проектирование. Структуризация сложных программных систем».

Приложение Стандарт кодирования для примеров кода

Для примеров кода в этой книге используется обобщенный объектно-ориентированный язык программирования. Его синтаксис представляет собой смесь PHP и Java. Вот его свойства:

- Он строго типизирован. Для параметров и возвращаемых значений требуются явные возвращаемые типы:

```
public function foo(Bar bar): Baz
{
    // возвращает экземпляр `Baz`
}
```

- Параметры, свойства и возвращаемые типы допускают `null` в качестве значения, если добавить знак вопроса в конце:

```
public function foo(Bar? bar): Baz?
{
    // допустим экземпляр `Bar` или `null`

    // возвращает экземпляр `Baz` или `null`
}
```

- Если не добавить знак вопроса после типа параметра или возвращаемого значения, значение `null` уже будет неприемлемо:

```
public function foo(Bar bar): Baz
{
    // bar представляет собой экземпляр `Bar`, не `null`

    // необходимо вернуть экземпляр `Baz`
}
```

- `void` — это специальный возвращаемый тип, который может использоваться, когда функция ничего не возвращает:

```
public function bar(): void
{
    // ничего не возвращается
}
```

- Поддерживаемые типы — это имена классов, примитивы (`string`, `int`, `float`, `bool`), массивы (`array`), вызываемые объекты (`callable`) и универсальные объекты (`object`):

```
public function foo(Bar bar, string baz): callable
{
    // вернуть вызываемый объект
}
```

- Вызываемые объекты — это функции, которые могут быть переданы другой функции:

```
// код запускает вызываемый объект, возвращенный `foo()`:
this.foo()();
```

- Public-метод объекта может быть передан как вызываемый:

```
eventDispatcher.addListener([object, 'methodName']);
```

- Метод конструктора объекта всегда называется `__construct()`. Классы могут быть помечены как `final`, чтобы сделать невозможным их расширение. В методах объекта `this` — это ссылка на объект, для которого вызывается метод:

```
final class Foo
{
    private string foo;

    public function __construct(string foo)
    {
        this.foo = foo;
    }
}
```

- Конструктор будет вызываться *во время* инстанцирования экземпляра объекта, а не после, это означает, что исключение внутри конструктора прервет инстанцирование экземпляра и приведет к возвращению `null`:

```
try {
    // Когда конструктор выдает исключение...
    foo = new Foo();
} catch (Exception exception) {
    //`foo` будет равно `null`
}
```

- Методы и свойства могут иметь область действия `private`, `protected` или `public`. Область относится к классу, а не к объекту, поэтому любой объект имеет доступ к приватным свойствам и методам любого другого объекта *того же типа*:

```
final class Foo
{
    private string foo;

    public function equals(Foo other): bool
    {
        return this.foo == other.foo;
    }
}
```

- Интерфейс (interface) определяет набор общедоступных методов, не предоставляя для них реализации:

```
interface Foo
{
    public function bar(Baz baz): string;
}

final class FooBar implements Foo
{
    public function bar(Baz baz): string
    {
        return 'Hello, world!';
    }
}
```

- Язык поддерживает вывод типа, что означает, что тип переменной является необязательным, если он может быть получен из значения, которое ему присваивается:

```
final class Foo
{
    private foo;

    public function __construct(string foo)
    {
        /*
        * Известно, что `foo`является `string`, поэтому
        * свойство `foo`не нужно помечать
        * как `string`.
        */
        this.foo = foo;

        /**
         * Известно, что `foo`– это `string`, поэтому при
         * присваивании его переменной `bar`
         * не нужно указывать тип.
```

```
         */
        bar = foo;
    }
}
```

- Тип массива будет вести себя как список или карта в зависимости от того, как его использовать:

```
list = [
    'foo',
    'bar'
];
// добавить еще один элемент в `list`:
list[] = 'baz';

map = [
    'foo' => 20,
    'bar' => 30
];
// добавить еще один элемент в `map`:
map['baz'] = 40;
```

- Классы имеют пространство имен, и можно импортировать классы из других пространств имен с помощью инструкций use:

```
namespace Namespace\Subnamespace\Etc;

use From\Other\Namespace\Bar;

final class Foo
{
    public function __construct(Bar bar)
    {
    }
}
```

- Объекты имеют волшебную константу, которая представляет полное имя класса объекта в виде строки:

```
// Это будет `foo`
foo.className
```

- Язык имеет стандартную библиотеку, которая предлагает такие функции, как strpos(), file_get_contents() и json_encode(), а также глобальные константы для влияния на поведение этих функций:

```
originalJsonData = file_get_contents('/path/to/file.json');
decodedData = json_decode(originalJsonData);

jsonDataEncodedAgain = json_encode(
    decodedData,
    JSON_THROW_ON_ERROR | JSON_FORCE_OBJECT?);
```

- Можно сравнивать значения с помощью оператора ==, который учитывает тип сравниваемых значений. Если сравниваются объекты, == будет иметь значение true только в том случае, если значения относятся к одному и тому же объекту:

```
'a' == 'a'; // true
'a' == 1; // error
new Foo() == new Foo(); // false

foo = new Foo();
bar = foo;
foo == bar; // true
```

- Можно вызвать любое исключение, которое остановит выполнение кода. Можно восстанавливаться после исключений, перехватывая их. Встроенные классы исключений могут быть расширены:

```
try {
// ...

    выдать новое исключение RuntimeException('Message');

    // это не будет выполнено
} catch (Exception exception) {
    // сделать что-нибудь с исключением, при желании
}

final class CustomException extends RuntimeException
{
    // ...
}
```

- Можно создать копию объекта с помощью оператора clone:

```
foo = new Foo();
copy = clone foo;
```

Маттиас Нобак

Объекты. Стильное ООП

Перевели с английского С. Черников, Р. Чикин

Руководитель дивизиона	*Ю. Сергиенко*
Ведущий редактор	*Е. Строганова*
Литературный редактор	*М. Трусковская*
Художественный редактор	*В. Мостипан*
Корректоры	*Т. Никифорова, Г. Шкатова*
Верстка	*Л. Егорова*

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

Дата изготовления: 02.2023. Наименование: книжная продукция. Срок годности: не ограничен.

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

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

Подписано в печать 26.12.22. Формат 70×100/16. Бумага офсетная. Усл. п. л. 24,510. Тираж 700. Заказ 0000.