Принципы разработки программных пакетов

Маттиас Нобак

Принципы разработки программных пакетов

ДМК ПРЕСС
издательство

Маттиас Нобак

Принципы разработки програмных пакетов

Проектирование повторно используемых компонентов

Principles of Package Design

Creating Reusable Software Components

Matthias Noback

Принципы разработки программных пакетов

Проектирование повторно используемых компонентов

Маттиас Нобак

Москва, 2020

УДК 004.971
ББК 32.972
Н72

Маттиас Нобак

Н72 Принципы разработки программных пакетов: Проектирование повторно используемых компонентов / пер. с анг. Д. А. Беликова. – М.: ДМК Пресс, 2020. – 274 с.: ил.

ISBN 978-5-97060-793-0

Существует масса литературы и онлайн-ресурсов, посвященных дизайну классов, но информацию о проектировании программных пакетов найти не так просто. Книга Маттиаса Нобака, профессионального PHP-разработчика, призвана восполнить этот пробел. В ней рассказывается о принципах повторного использования и распространения компонентов, также известных как пакеты, и предлагается ряд полезных техник по организации кода в группы любого размера. Вы узнаете о том, какие классы должны быть внутри пакета, как использовать принципы связности и зацепления, как облегчить поддержку пакета.

Издание адресовано программистам, использующим объектно-ориентированный язык для создания приложений. Представленные в книге примеры кода поясняют отдельные технические моменты и упрощают понимание материала.

УДК 004.971
ББК 32.972

All rights reserved. This work may not be translated or copied in whole or in part without the written permission of the author, except for brief excerpts in connection with reviews or scholarly analysis. Use in connection with any form of information storage and retrieval, electronic adaptation, or computer software is forbidden

Все права защищены. Любая часть этой книги не может быть воспроизведена в какой бы то ни было форме и какими бы то ни было средствами без письменного разрешения владельцев авторских прав.

ISBN 978-1-4842-4118-9 (анг.)
ISBN 978-5-97060-793-0 (рус.)

© 2019 Matthias Noback
© Оформление, издание, перевод, ДМК Пресс, 2020

Оглавление

Предисловие от издательства ... **11**

Об авторе ... **12**

О техническом рецензенте ... **13**

Благодарности ... **14**

Введение ... **16**

ЧАСТЬ I. ПРОЕКТИРОВАНИЕ КЛАССОВ ... **21**

Глава 1 . Принцип единственной ответственности ... **25**

Класс со множественными ответственностями ... 25

Ответственности порождают причины для изменения ... 27

Рефакторинг: использование взаимодействующих классов ... 28

Преимущества единственной ответственности ... 29

Пакеты и принцип единственной ответственности ... 30

Заключение ... 31

Глава 2. Принцип открытости/закрытости ... **32**

Класс, который закрыт для расширения ... 32

Рефакторинг: абстрактная фабрика ... 35

Рефакторинг: делаем абстрактную фабрику открытой для расширения ... 39

Замена или декорирование фабрики кодировщика ... 40

Делаем EncoderFactory открытым для расширения ... 41

Рефакторинг: полиморфизм ... 44

Пакеты и принцип открытости/закрытости ... 47

Заключение ... 48

Глава 3. Принцип подстановки Барбары Лисков ... **50**

Нарушение: производный класс не имеет реализации всех методов ... 52

Протекающие абстракции ... 56

Нарушение: разные замены возвращают вещи разных типов ... 57

Допустимы более конкретные типы возвращаемых значений 61
Нарушение: производный класс менее снисходителен касательно аргументов метода 62
Нарушение: тайное программирование более специфического типа 66
Пакеты и принцип подстановки Барбары Лисков 69
Заключение 70
Глава 4. Принцип разделения интерфейса 72
Нарушение: многократные варианты использования 72
Рефакторинг: отдельные интерфейсы и множественное наследование 74
Нарушение: никакого интерфейса, просто класс 77
Неявные изменения в неявном интерфейсе 78
Рефакторинг: добавление интерфейсов заголовка и роли 79
Пакеты и принцип разделения интерфейса 81
Заключение 82
Глава 5. Принцип инверсии зависимостей 83
Пример инверсии зависимостей: генератор FizzBuzz 83
Делаем класс FizzBuzz открытым для расширения 85
Избавляемся от специфичности 86
Нарушение: высокоуровневый класс зависит от низкоуровнего 89
Рефакторинг: абстракции и конкретные реализации зависят от абстракций 92
Нарушение: привязка к поставщику 96
Решение: добавляем абстракцию и удаляем зависимость с помощью композиции 100
Пакеты и принцип инверсии зависимостей 103
Зависимость от стороннего кода: всегда ли это плохо? 104
Когда публиковать явный интерфейс класса 106
Если не все открытые методы предназначены для использования обычными клиентами 107
Если класс использует ввод/вывод 108
Если класс зависит от стороннего кода 109

Если вы хотите ввести абстракцию для множества конкретных вещей 111
Если вы предвидите, что пользователь захочет заменить часть иерархии объектов........ 112
Для всего остального: придерживайтесь финального класса 114
Заключение 115

ЧАСТЬ II. РАЗРАБОТКА ПАКЕТОВ........ 117

Глава 6. Принцип эквивалентности повторного использования и выпуска........ 127

Держите свой пакет в системе управления версиями 129
Добавьте файл определений........ 129
Семантическое версионирование 130
Проектирование для обратной совместимости 132
Практические правила 133
Ничего не выбрасывайте 134
Когда вы что-то переименовываете, добавьте посредника........ 134
Добавляйте параметры только в конце и со значением по умолчанию 136
Методы не должны иметь скрытых побочных эффектов........ 137
Версии зависимостей не должны быть очень строгими........ 138
Используйте объекты вместо значений примитивного типа........ 139
Используйте объекты для инкапсуляции состояния и поведения 142
Используйте фабрики объектов 144
И так далее… 145
Добавьте метафайлы 146
Файл README и документация........ 146
Установка и настройка 147
Применение........ 147
Точки расширения (не обязательно)........ 147
Ограничения (не обязательно) 147
Лицензия........ 147

Журнал изменений (не обязательно) 148
Примечания касательно обновлений (не обязательно) 149
Руководство по содействию (не обязательно) 150
Контроль качества 150
Качество с точки зрения пользователя 150
Что нужно сделать человеку, отвечающему за поддержку пакета 152
Статический анализ 152
Добавьте тесты 152
Настройте непрерывную интеграцию 153
Заключение 154
Глава 7. Принцип совместного повторного использования 156
Пласты функций 158
Очевидное расслоение 158
Скрытое расслоение 160
Классы, которые можно использовать, только когда установлен… 162
Предлагаемый рефакторинг 166
Пакет должен быть «компонуемым» 167
Чистые релизы 169
Бонусные функции 173
Предлагаемый рефакторинг 175
Наводящие вопросы 177
Когда применять этот принцип 178
Когда нарушать принцип 179
Почему не следует нарушать этот принцип 179
Заключение 180
Глава 8. Принцип общей закрытости 181
Изменение в одной из зависимостей 182
Assetic 183
Изменение на уровне приложения 185
FOSUserBundle 186
Изменения, продиктованные бизнесом 189
Sylius 190
Бизнес-логика 191

Треугольник принципов связности 193
Заключение 195

Глава 9. Принцип ацикличности зависимостей 196

Зацепление: выявление зависимостей 196
Различные способы зацепления пакетов 197
Композиция 198
Наследование 198
Реализация 199
Использование 199
Инстанцирование 199
Использование глобальной функции 200
Что не следует учитывать: глобальное состояние 201
Визуализация зависимостей 201
Принцип ацикличности зависимостей 203
Проблемные циклы 204
Решения, позволяющие разорвать циклы 207
Псевдоциклы и избавление от них 207
Реальные циклы и избавление от них 210
Инверсия зависимостей 211
Инверсия управления 213
Посредник 214
Цепочка обязанностей 217
Сочетание Посредника и Цепочки обязанностей: система событий 218
Заключение 223

Глава 10. Принцип устойчивых зависимостей 224

Устойчивость 225
Не каждый пакет может быть высокостабильным 228
Нестабильные пакеты должны зависеть только от более стабильных 229
Оценка устойчивости 230
Снижение нестабильности и повышение устойчивости 231
Вопрос: следует ли учитывать все пакеты, какие есть во вселенной? 233
Нарушение: ваш стабильный пакет зависит от стороннего нестабильного пакета 234

Решение: используйте инверсию зависимостей 237
Пакет может быть как ответственным, так и наоборот 240
Заключение 242
Глава 11. Принцип устойчивых абстракций 243
Устойчивость и абстрактность 243
Как определить, является ли пакет абстрактным 246
A-метрика 246
Абстрактные вещи в стабильных пакетах 247
Абстрактность возрастает по мере роста устойчивости 248
Главная последовательность 249
Типы пакетов 251
Странные пакеты 252
Заключение 254
Глава 12. Заключение 256
Создание пакетов – это сложно 256
Повторное использование «в малом» 256
Повторное использование «в большом» 257
Разнородность программного обеспечения 258
Повторное использование компонентов возможно, но требует больше работы 259
Создание пакетов – выполнимая задача 259
Уменьшение воздействия первого из трех правил 260
Уменьшение воздействия второго из трех правил 261
Создание пакетов – это легко? 262
Приложение А. Полный вариант класса Page 263
Предметный указатель 273

Предисловие от издательства

Отзывы и пожелания

Мы всегда рады отзывам наших читателей. Расскажите нам, что вы думаете об этой книге – что понравилось или, может быть, не понравилось. Отзывы важны для нас, чтобы выпускать книги, которые будут для вас максимально полезны.

Вы можете написать отзыв прямо на нашем сайте **www.dmkpress.com**, зайдя на страницу книги, и оставить комментарий в разделе «Отзывы и рецензии». Также можно послать письмо главному редактору по адресу **dmkpress@gmail.com**, при этом напишите название книги в теме письма.

Если есть тема, в которой вы квалифицированы, и вы заинтересованы в написании новой книги, заполните форму на нашем сайте по адресу **http://dmkpress.com/authors/publish_book/** или напишите в издательство по адресу **dmkpress@gmail.com**.

Список опечаток

Хотя мы приняли все возможные меры для того, чтобы удостовериться в качестве наших текстов, ошибки все равно случаются. Если вы найдете ошибку в одной из наших книг – возможно, ошибку в тексте или в коде, – мы будем очень благодарны, если вы сообщите нам о ней. Сделав это, вы избавите других читателей от расстройств и поможете нам улучшить последующие версии этой книги.

Если вы найдете какие-либо ошибки в коде, пожалуйста, сообщите о них главному редактору по адресу **dmkpress@gmail.com**, и мы исправим это в следующих тиражах.

Нарушение авторских прав

Пиратство в интернете по-прежнему остается насущной проблемой. Издательство «ДМК Пресс» очень серьезно относится к вопросам защиты авторских прав и лицензирования. Если вы столкнетесь в интернете с незаконно выполненной копией любой нашей книги, пожалуйста, сообщите нам адрес копии или веб-сайта, чтобы мы могли применить санкции.

Пожалуйста, свяжитесь с нами по адресу электронной почты **dmkpress@gmail.com** со ссылкой на подозрительные материалы.

Мы высоко ценим любую помощь по защите наших авторов, помогающую нам предоставлять вам качественные материалы.

Об авторе

Маттиас Нобак – профессиональный PHP-разработчик. Он является основателем компании Noback's Office, специализирующейся на веб-разработке, тренингах и консалтинге. Ранее он работал в качестве разработчика в компании Driebit (Амстердам) и IPPZ (Утрехт), а также в качестве технического директора (CTO) в компании Ibuildings (Утрехт). С 2011 года он ведет блог (matthiasnoback.nl), где пишет на темы, относящиеся к продвинутым аспектам разработки программного обеспечения. Более всего Маттиаса интересует архитектура программного обеспечения, унаследованный код, тестирование и дизайн объектов в ООП. Маттиас также является автором книг «Один год с Symfony« и «Микросервисы для всех». В Twitter Маттиас пишет под ником @matthiasnoback.

О техническом рецензенте

Росс Так (Ross Tuck) – инженер по разработке программного обеспечения и тренер. На конференциях по всему миру он рассказывает о разработке программного обеспечения, а также является участником подкастов, автором статей и время от времени вносит свой вклад в репозитории на GitHub. Будучи родом из США, сейчас он проживает в Нидерландах вместе супругой и котом. В Twitter он известен как @rosstuck.

Благодарности

В первую очередь хочу выразить благодарность нескольким людям.

Во-первых, Роберту Мартину (Robert C. Martin). Многие принципы проектирования, рассмотренные в этой книге, были рождены в результате его работы.

Также хочу поблагодарить тех, кто предоставил ценные замечания в процессе корректуры первой версии книги, а именно: Брайан Фентон (Brian Fenton), Кевин Арчер (Kevin Archer), Луис Кордова (Luis Cordova), Марк Бадолато (Mark Badolato), Маттис ван Ритшотен (Matthijs van Rietschoten), Рамон де ла Фуэнте (Ramon de la Fuente), Ричард Перез (Richard Perez), Рольф Бабийн (Rolf Babijn), Себастиан Сток (Sebastian Stok), Томас Стоун (Thomas Shone) и Питер Рем (Peter Rehm).

Одного из корректоров я хочу упомянуть отдельно – это Питер Бойер (Peter Bowyer), он был автором многих детальных предложений. Он также сделал одно блестящее предложение – сделать эту книгу, изначально ориентированную на PHP, интересной и полезной любому разработчику. И хотя эта книга не привязана к конкретному языку программирования, я многим обязан одному конкретному сообществу, к которому я принадлежу: сообществу PHP-разработчиков.

Спасибо вам всем за то, что вы такие классные – каждый день предоставляете отличные материалы для чтения, генерируете замечательные идеи, приглашаете по-дружески на различные мероприятия, пишите много хорошего кода. Огромное спасибо Россу (Ross Tuck), который выполнил тщательный технический обзор второй редакции этой книги. И хотя он добавил мне много работы, я очень рад, что он это сделал. Он сделал много рекомендаций по тому, как я мог бы сделать книгу более полезной и более понятной для широкого круга читателей с разными точками зрения и разным опытом в этой области.

Отдельное спасибо издательству Apress, в частности Шиве (Shiva), Лауре (Laura) и Рите (Rita) – за то, что взялись за работу

над книгой «Принципы разработки пакетов», и за то, что предоставили возможность выпустить ее в значительно лучшей форме, в то же время делая ее доступной для значительно большего числа читателей.

И наконец, спасибо вам, Лайз (Lies), Лукас (Lucas) и Джулия (Julia). Спасибо вам, что позволили мне отвлечься от семейного бизнеса и написать эту книгу в одиночестве. И спасибо, что всегда обнимали меня, когда я возвращался.

Введение

В процессе написания я полагал что вы, читающие эту книгу, программисты и используете объектно-ориентированный язык для создания приложений. Это означает, что у вас есть некоторый опыт в создании классов, методов, интерфейсов и т. д. и вы так или иначе пытаетесь заставить все эти элементы корректно работать вместе. И хотя вы, вероятно, знаете, как это сделать, время от времени вы задаетесь вопросом: «А правильно ли я всё делаю?»

Это хороший и понятный вопрос. Будучи программистом, вы ежедневно вынуждены принимать множество решений – вполне естественно беспокоиться о том, принимаются ли правильные решения. Плохое решение сегодня может превратиться в большие объемы дополнительной работы позже.

К сожалению, нет простого способа понять, делаете ли вы что-то правильно. Лучшее, что можно сделать, – это следить за развитием событий и быть готовым изменить курс, если это будет необходимо. Но чтобы тренировать внимание и способность предсказывать, как будут развиваться события, нужно, помимо собственного опыта, также обращаться и к другим источникам. Например, можно читать книги по программированию или же перенимать опыт других программистов.

Когда я хочу выложить какой-то код в open source, полагая, что он может быть полезен другим разработчикам, я тоже задаюсь вопросом «а делаю ли я это правильно?». И я начал искать источники знаний, где мог бы получить ответ на свой вопрос. Что касается дизайна классов, существует огромное количество онлайн- и офлайн-ресурсов. Намного больше, чем ресурсов о дизайне программных пакетов. Я не смог найти много материала, который помог бы мне лучше создавать пакеты, за исключением нескольких разделов в книге (и никогда целой книги, посвященной этому вопросу!) и нескольких старых статей.

Одним из часто встречающихся источников был веб-сайт Роберта Мартина (Robert C. Martin's) butunclebob.com, на котором есть несколько статей о принципах проектирования SOLID и две статьи о принципах проектирования компонентов. В этих статьях Роберт

предлагает несколько очень простых принципов проектирования повторно используемых компонентов. Когда я впервые прочел данный материал, мне тут же стало ясно, что каждый программист должен знать эти принципы. Так я начал писать эту книгу, развивая принципы «дяди Боба» и разъясняя их в контексте создания пригодных для повторного использования и распространения компонентов, также известных как «пакеты». Книга «Принципы проектирования пакетов» предоставляет ответы на следующие вопросы:

- какие классы должны быть внутри пакета, а какие нет?
- какие зависимости опасны, а какие нет?
- как сделать использование пакета комфортнее для пользователя?
- как облегчить поддержку пакета?

Если вы заинтересованы в создании ваших собственных программных пакетов (не обязательно с открытым исходным кодом), ответы на эти вопросы помогут вам начать делать это правильно с самого начала. Эти принципы будут направлять вас по ходу разработки. Если вы уже создавали пакеты ранее, знание этих принципов поможет вам улучшить их в следующих релизах.

С другой стороны, если вы не заинтересованы в разработке пакетов, из этой книги вы все равно почерпнете полезные знания. Это возможно, во-первых, потому что эта книга содержит много подсказок по хорошему дизайну классов. Во-вторых, потому что принципы проектирования пакетов позволят вам лучше структурировать любой проект по разработке программного обеспечения, будь то библиотека для многоразового использования, какой-то отдельный компонент или же целый модуль в составе приложения. Эта книга предлагает много полезных техник по организации вашего кода в группы любого размера.

Обзор содержания

Большая часть этой книги освещает принципы создания программных пакетов. Однако в первую очередь мы должны определиться с составом пакета: классами и интерфейсами. То, как вы выполните их дизайн, будет иметь огромное влияние на характеристики пакета, в котором они в конечном итоге будут расположены. Таким образом, перед тем как начать рассмотрение собственно принципов проектирования пакетов, нам необходимо рассмотреть принципы дизайна классов. Эти принципы также известны как SOLID.

Каждая буква данного акронима отвечает за свой принцип, и мы кратко рассмотрим их в первой части этой книги.

Вторая часть книги охватывает шесть главных принципов проектирования пакетов. Первые три из них относятся к связности. В то время как связность классов освещает вопросы о том, какие методы должны принадлежать классу, связность пакетов же повествует о том, какие классы должны принадлежать конкретному пакету. Принципы связности пакетов расскажут вам о том, какие классы следует объединять в пакет, когда следует разбивать пакет на несколько и, в первую очередь, в каких случаях комбинация классов может считаться «пакетом».

Следующие три принципа проектирования пакетов касаются зацепления. Зацепление важно на уровне классов, так как большинство классов бесполезны сами по себе. Им требуются другие классы, с которыми они взаимодействуют. Принципы проектирования классов, такие как инверсия зависимостей, помогут вам писать хорошие независимые классы. Но в случае когда зависимости класса живут вне пакета, которому этот класс принадлежит, вам потребуется способ определения того, безопасно ли такое зацепление пакетов между собой. Принципы зацепления пакетов помогут вам выбрать правильные зависимости. Они также помогут вам распознать и предотвратить ошибки на графе зависимостей ваших пакетов.

О примерах кода

Несмотря на то что я хотел бы, чтобы эта книга была бы полезна любому программисту, примеры кода так или иначе должны быть написаны на каком-то языке программирования. Я выбрал PHP, потому что этот язык я знаю лучше всего и он наиболее удобен для меня. Если вы не знаете PHP, это не должно стать проблемой в понимании кода, при условии что вы знакомы с любым другим объектно-ориентированным языком программирования.

Помните, что примеры кода из книги не готовы для использования в реальных приложениях. Они предназначены лишь для того, чтобы донести до читателей некоторые технические моменты. Ни в коем случае не копируйте их в свои проекты «как есть».

Чтобы упростить примеры кода и выделить наиболее важные области, я разработал следующие соглашения.

- Я сокращаю объявления свойств и методов, используя // …
- Я сокращаю выражения, используя …
- Когда требуется показать код, упоминавшийся ранее, но измененный, я повторяю как можно меньше исходного кода.
- Хотя я не считаю целесообразным использовать суффикс «Interface», я все же использую его в примерах кода, поскольку он упрощает обсуждение интерфейсов в обычном тексте.
- Хотя я считаю, что лучше всего объявлять классы как «Final» (позже будет объяснено, почему), я не буду делать это в большинстве примеров кода, потому что это может немного отвлекать.

ЧАСТЬ I

ПРОЕКТИРОВАНИЕ КЛАССОВ

Разработчики, как вы и я, нуждаются в помощи при принятии решений, так как мы принимаем их множество, каждый день, день за днем. Так что если есть какие-то принципы, которые мы считаем обоснованными, мы с радостью следуем им. Принципы являются методическими рекомендациями, теми «вещами, что нужно делать». Однако же они не являются строгими правилами. Вы не обязаны следовать этим принципам, но если быть откровенным – все-таки лучше следовать им.

Когда дело доходит до создания классов, есть много рекомендаций, которым вы должны следовать, таких как: выбирать описательные имена, не использовать много переменных, использовать как можно меньше управляющих структур и т. д. Но на самом деле это довольно общие рекомендации по программированию. Они помогут поддерживать ваш код читаемым, понимаемым и, как следствие, поддерживаемым. Кроме того, они довольно специфичны, поэтому ваша команда может быть очень строга к ним («не более двух уровней отступов внутри каждого метода», «не более трех переменных экземпляров» и т. д.).

Наряду с этими общими руководящими принципами программирования существуют также более глубокие принципы, которые могут быть применены к дизайну классов. Каждый из этих принципов дает простор для обсуждения. Кроме того, не все из них могут или должны применяться постоянно (в отличие от более общих рекомендаций по программированию – когда они не применяются, ваш код наверняка начнет мешать вам очень скоро).

Принципы, на которые я ссылаюсь, называются SOLID – это имя дано им Робертом Мартином (Robert Martin). В следующих главах я кратко излагаю каждый из этих принципов. Несмотря на то что принципы SOLID связаны с дизайном классов, их обсуждение относится к этой книге, поскольку принципы дизайна классов соответствуют принципам дизайна программных пакетов, которые мы обсудим во второй части этой книги.

Зачем следовать принципам?

Когда вы узнаете о принципах SOLID, вы можете спросить себя: почему я должен следовать им? Возьмем, к примеру, принцип открытости/закрытости: «Вы должны иметь возможность расширять поведение класса, не изменяя его». Почему, собственно, я должен это делать? Неужели изменить поведение класса, открыв его файл в редакторе и внеся некоторые изменения, – это плохая практика? Или возьмем, к примеру, принцип инверсии зависимостей, который гласит: «Необходимо зависеть от абстракций, а не от конкретных реализаций». Опять же, почему? Что не так в зависимостях от конкретных реализаций?

В следующих главах я, конечно же, приложу все свои силы для того, чтобы объяснить вам, почему вы должны использовать эти принципы и что произойдет, если вы проигнорируете их. Но прежде чем вы погрузитесь глубже, я хочу кое-что пояснить: принципы SOLID применяются в дизайне классов для того, чтобы подготовить ваш код к дальнейшим изменениям. Вы ведь хотите, чтобы эти изменения были локальными, а не глобальными и как можно более мелкими.

Готовимся к изменениям

Почему вы хотите делать как можно меньше и как можно более мелких изменений в существующем коде? Во-первых, всегда есть риск, что одно из этих изменений сломает систему целиком. Также любое изменение существующего класса требует некоторого количества времени – нужно понять, что этот класс изначально делал и где лучше всего добавить или удалить несколько строк кода. Также требуется дополнительно модифицировать существующие модульные тесты для изменяемого класса. Кроме того, каждое изменение может быть включено в некоторый процесс рецензирования. Также, возможно, потребуется повторная сборка всей систе-

мы, или даже будет необходимо другим командам обновить свои системы, чтобы учесть ваши изменения.

Все эти сложности могли бы привести нас к заключению, что не следует изменять существующий код. Тем не менее совсем избегать изменений мы не можем. Большинство реальных бизнесов находятся в процессе постоянного изменения, что, в свою очередь, приводит к изменениям в требованиях к программному обеспечению. Таким образом, чтобы продолжать работать в качестве разработчика программного обеспечения, вы должны смириться с изменениями. И чтобы вам было легче справляться с быстро меняющимися требованиями, вам необходимо подготовить свой код для них. К счастью, для этого есть много различных приемов, которые можно узнать из следующих пяти принципов дизайна классов – SOLID.

Глава 1

Принцип единственной ответственности

Принцип единственной ответственности гласит[1]:

Класс должен иметь одну и только одну причину для изменения.

При первом знакомстве с этим принципом может показаться странным, что сначала говорится об «ответственности», а потом о «причине для изменения». Но если задуматься, это не так странно – каждая ответственность также является и причиной для изменения.

Класс со множественными ответственностями

Давайте рассмотрим конкретный, вероятно, узнаваемый многими из вас, пример класса, который используется для отправки подтверждения на адрес электронной почты нового пользователя (см. листинг 1-1 и рис. 1-1). У него есть некоторые зависимости, такие как шаблонизатор для рендеринга тела сообщения, сервис-переводчик для перевода темы сообщения и почтовый клиент для отправки сообщения. Все они внедряются в класс по интерфейсам (что само по себе правильно; *см.* главу 5).

Листинг 1-1. Класс ConfirmationMailMailer

```
class ConfirmationMailMailer {
private $templating;
private $translator;
```

[1] *Robert C. Martin.* The Principles of OOD // http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

```
private $mailer;
public function __construct(
TemplatingEngineInterface $templating,
TranslatorInterface $translator,
MailerInterface $mailer
) {
$this->templating = $templating;
$this->translator = $translator;
$this->mailer = $mailer;
}
public function sendTo(User $user): void {
$message = $this->createMessageFor($user);
$this->sendMessage($message);
}
private function createMessageFor(User $user): Message {
$subject = $this ->translator ->translate(
'Confirm your mail address'
);
$body = $this->templating ->render(
'confirmationMail.html.tpl', [
'confirmationCode' => $user->getConfirmationCode()
]
);
$message = new Message($subject, $body);
$message->setTo($user->getEmailAddress());
return $message;
}
private function sendMessage(Message $message): void {
$this->mailer->send($message);
}
}
```

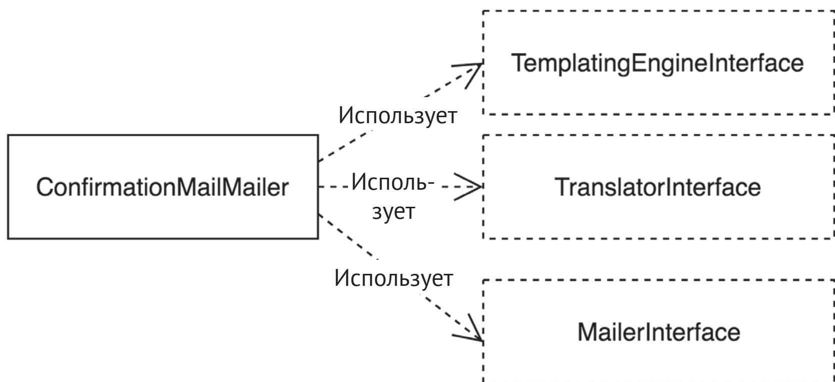


Рис. 1-1. Диаграмма исходной ситуации

Ответственности порождают причины для изменения

Если бы вы беседовали с кем-то об этом классе, вы бы сказали, что у него есть два задания, или две ответственности, – *создать* письмо с подтверждением и *отправить* его. Эти две ответственности также представляют собой две причины для изменений. Всякий раз, когда изменяются требования, касающиеся создания или отправки сообщения, этот класс необходимо будет модифицировать. Это также означает, что когда любая из ответственностей требует изменения, *весь* класс должен быть открыт и изменен, в то время как большая часть его может не иметь ничего общего с запрошенным изменением.

Поскольку изменение существующего кода – это то, что необходимо предотвратить или, по крайней мере, ограничить (*см.* введение), а ответственности – это причины для изменения, мы должны попытаться свести к минимуму количество ответственностей каждого класса. В то же время это минимизирует вероятность того, что класс должен быть открыт для модификации.

Поскольку класс, не имеющий ответственностей, бесполезен, лучшее, что мы можем сделать относительно минимизации числа ответственностей, – это сократить их до одной. Отсюда и принцип *единственной ответственности*.

НАРУШЕНИЯ ПРИНЦИПА ЕДИНСТВЕННОЙ ОТВЕТСТВЕННОСТИ

Ниже приводится список признаков класса, которые могут нарушать принцип *единственной ответственности*:

- у класса много переменных экземпляра;
- в классе много открытых методов;
- каждый метод класса использует разные переменные экземпляра;
- конкретные задачи делегируются закрытым методам.

Все это является вескими причинами для извлечения из класса так называемых «взаимодействующих классов», делегируя тем самым некоторые ответственности класса и заставляя его соответствовать принципу *единственной ответственности*.

Рефакторинг: использование взаимодействующих классов

Теперь мы знаем, что класс `ConfirmationMailMailer` делает слишком много вещей. Мы можем (и в этом случае должны) перепроектировать класс, извлекая взаимодействующие классы. Поскольку этот класс представляет собой «почтового клиента», мы оставляем за ним ответственность по *отправке сообщения* пользователю, но извлекаем ответственность за *создание сообщения*.

Создание сообщения немного сложнее, нежели простое инстанцирование объекта с использованием оператора `new`, и даже требует нескольких зависимостей. Здесь необходим специальный «фабричный» класс – `ConfirmationMailFactory` (*см.* листинг 1-2 и рис. 1-2).

Листинг 1-2. Класс ConfirmationMailFactory

```
class ConfirmationMailMailer
{
private $confirmationMailFactory;
private $mailer;
public function __construct(
ConfirmationMailFactory $confirmationMailFactory
MailerInterface $mailer
) {
$this->confirmationMailFactory = $confirmationMailFactory;
$this->mailer = $mailer;
}
public function sendTo(User $user): void
{
$message = $this->createMessageFor($user);
$this->sendMessage($message);
}
private function createMessageFor(User $user): Message
{
return $this->confirmationMailFactory
->createMessageFor($user);
}
private function sendMessage(Message $message): void
{
$this->mailer->send($message);
}
}
class ConfirmationMailFactory
{
private $templating;
```

```
private $translator;
public function __construct(
TemplatingEngineInterface $templating,
TranslatorInterface $translator
) {
$this->templating = $templating;
$this->translator = $translator;
}
public function createMessageFor(User $user): Message
{
/*
* Создаем экземпляр Message на основе данного пользователя;
*/
$message = ...;
return $message;
}
}
```

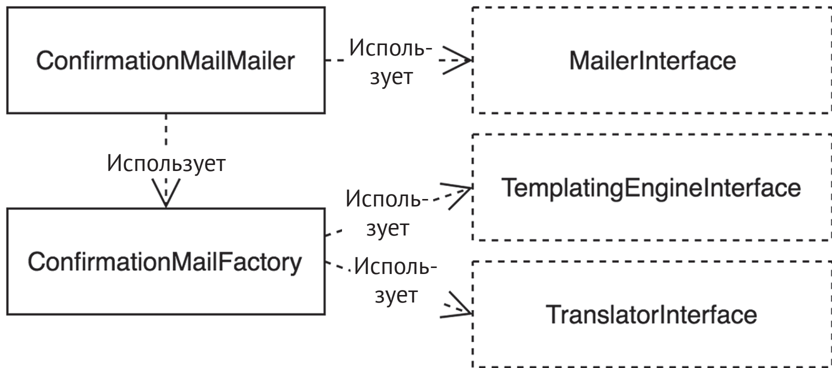


Рис. 1-2. Используем класс ConfirmationMailFactory

Теперь логика создания письма с подтверждением помещена в класс `ConfirmationMailFactory`. Было бы еще лучше, если бы для этого фабричного класса был определен интерфейс, но пока и этого будет достаточно.

Преимущества единственной ответственности

В качестве побочного эффекта такого рефакторинга можно упомянуть тот факт, что оба класса легче тестировать. Теперь вы можете протестировать обе ответственности по отдельности. Правильность созданного сообщения можно проверить, протестировав ме-

тод `createMessageFor()` из класса `ConfirmationMailFactory`. Протестировать метод `sendTo()` теперь также довольно просто, потому что вы можете мокировать весь процесс создания сообщения и просто сосредоточиться на отправке сообщения.

В целом вы заметите, что классы с единственной ответственностью легче тестировать. Единственная ответственность делает класс меньше, поэтому вам придется писать меньше тестов, чтобы охватить этот класс. Вам будет легче разобраться. Кроме того, в этих небольших классах будет меньше закрытых методов с эффектами, которые необходимо проверить в модульном тесте.

Наконец, классы меньшего размера также проще поддерживать. Их назначение проще понять, а все детали реализации находятся там, где и должны быть: в ответственных за них классах.

Пакеты и принцип единственной ответственности

Хотя принцип *единственной ответственности* должен применяться к классам, несколько иным образом он также должен применяться и к группам классов (также известным как *пакеты*). В контексте пакетного проектирования фраза «иметь только одну причину для изменения» превращается в «быть закрытым от изменений такого же типа». Соответствующий пакетный принцип называется принципом *общей закрытости* (*см.* главу 8).

В качестве несколько преувеличенного примера пакета, который не следует этому принципу *согласованного изменения*, можно привести пакет, который знает, как соединиться с базой данных MySQL *и* как создавать HTML-страницы. У такого пакета будет слишком много ответственностей, и он будет открыт (т. е. изменен) по разным причинам. Решение для таких пакетов, как этот, состоит в том, чтобы разделить их на более мелкие пакеты, у каждого из которых будет меньшее количество ответственностей и, следовательно, меньше причин для изменения.

Существует еще одно интересное сходство между принципом *единственной ответственности* при проектировании классов и принципом *общей закрытости* при проектировании пакета, о котором я хотел бы здесь быстро упомянуть: следование этим принципам в большинстве случаев уменьшает зацепление классов (и пакетов).

Когда у класса много *ответственностей*, у него также может быть много *зависимостей*. Вероятно, он получает много объектов, вводимых в качестве аргументов конструктора, чтобы иметь возможность выполнить свою цель. Например, классу `ConfirmationMailMailer` для создания и отправки письма с подтверждением потребовался сервис-переводчик, шаблонизатор и почтовый клиент. Находясь в зависимости от этих объектов, он был напрямую связан с ними. Когда мы применили принцип *единственной ответственности* и перенесли ответственность за создание сообщения в новый класс с именем `ConfirmationMailFactory`, мы сократили число зависимостей класса `ConfirmationMailMailer` и тем самым уменьшили его зацепление.

То же самое касается принципа *согласованного изменения*. Когда у пакета много зависимостей, он тесно связан с каждой из них, а это означает, что изменение в одной из зависимостей, вероятно, также потребует и изменения в пакете. Применение принципа *общей закрытости* к пакету означает уменьшение количества причин для его изменения. Удаление зависимостей или перенос их в другие пакеты – один из способов сделать это.

Заключение

У каждого класса есть ответственности, то есть вещи, которые он должен сделать. Ответственности также являются причинами для изменений. Принцип *единственной ответственности* велит нам ограничить количество обязанностей каждого класса, чтобы свести к минимуму количество причин для изменения класса.

Ограничение количества ответственностей обычно приводит к извлечению одного или нескольких взаимодействующих классов. Каждый из этих классов будет иметь меньшее количество зависимостей. Это полезно для разработки пакетов, поскольку каждый класс будет легче создавать, тестировать и использовать.

Глава 2

Принцип открытости/закрытости

Принцип открытости/закрытости гласит[1]:

> *Вы должны иметь возможность расширять поведение класса, не изменяя его.*

Опять же, необходимо сделать небольшой лингвистический переход от названия принципа к его объяснению: единицу кода можно считать «открытой для расширения», когда ее поведение можно легко изменить, *не* изменяя сам код. Тот факт, что для изменения поведения единицы кода не требуется никакой реальной модификации, делает ее «закрытой» для модификации.

Следует отметить, что возможность расширять поведение класса не означает, что вы действительно должны *расширять* этот класс, создав для него подкласс. Расширение класса означает, что вы можете влиять на его поведение извне, не касаясь класса или иерархии классов.

Класс, который закрыт для расширения

Взгляните на класс `GenericEncoder`, показанный в листинге 2-1 и на рис. 2-1. Обратите внимание на ветвление внутри метода `encodeToFormat()`, которое необходимо для выбора правильного кодировщика на основе значения аргумента `$format`.

---

1 *Robert C. Martin*. The Principles of OOD // http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

Листинг 2-1. Класс GenericEncoder

```
class GenericEncoder
{
public function encodeToFormat($data, string $format): string
{
if ($format === 'json') {
$encoder = new JsonEncoder();
} elseif ($format === 'xml') {
$encoder = new XmlEncoder();
} else {
throw new InvalidArgumentException('Unknown format');
}
$data = $this->prepareData($data, $format);
return $encoder->encode($data);
}
}
```

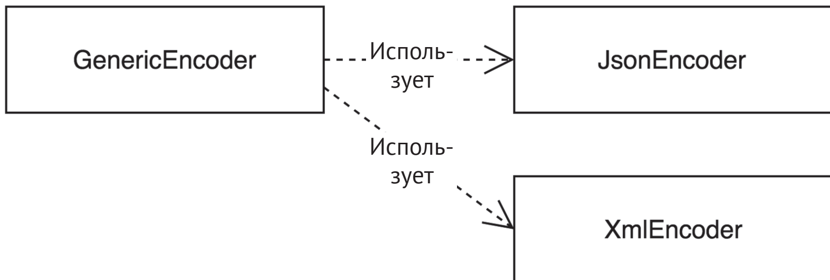


Рис. 2-1. Исходная ситуация

Допустим, вы хотите использовать класс `GenericEncoder` для кодирования данных в формат Yaml, который в настоящее время не поддерживается кодировщиком. Очевидным решением будет создать для этой цели класс `YamlEncoder`, а затем добавить дополнительное условие в существующий метод `encodeToFormat()`, приведенный в листинге 2-2.

Листинг 2-2. Добавление другого формата кодировки

```
class GenericEncoder
{
public function encodeToFormat($data, string $format): string
{
if (...) {
// ...
} elseif (...) {
```

```
// ...
} elseif ($format === 'yaml') {
$encoder = new YamlEncoder();
} else {
// ...
}
// ...
}
}
```

Как вы можете себе представить, каждый раз, когда вы хотите добавить еще один кодировщик для конкретного формата, необходимо менять сам класс `GenericEncoder`: нельзя изменить его поведение, не изменив его код. Вот почему этот класс нельзя рассматривать как *открытый для расширения* и *закрытый для модификации*.

Давайте посмотрим на метод `prepareData()` из того же класса. Как и метод `encodeToFormat()`, он содержит более специфичную для формата логику (*см.* листинг 2-3).

Листинг 2-3. Метод prepareData()

```
class GenericEncoder
{
public function encodeToFormat($data, string $format): string
{
// ...
$data = $this->prepareData($data, $format);
// ...
}
private function prepareData($data, string $format)
{
switch ($format) {
case 'json':
$data = $this->forceArray($data);
$data = $this->fixKeys($data);
// fall through
case 'xml':
$data = $this->fixAttributes($data);
break;
default:
throw new InvalidArgumentException(
'Format not supported'
);
}
return $data;
}
}
```

Метод `prepareData()` – еще один хороший пример закрытого для расширения кода, поскольку добавить поддержку другого формата без изменения самого кода *невозможно*. Кроме того, такого рода операторы `switch` не годятся для удобства обслуживания. Если вам придется изменить этот код, например при вводе нового формата, вполне вероятно, что вы либо используете некое дублирование кода, либо просто сделаете ошибку, потому что пропустили оператор `case`.

КЛАССЫ, НАРУШАЮЩИЕ ПРИНЦИП ОТКРЫТОСТИ/ЗАКРЫТОСТИ

Это список характеристик класса, который не может быть открыт для расширения:

- класс содержит условия для определения стратегии;
- условия, использующие одни и те же переменные или константы, повторяются внутри класса или связанных классов;
- класс содержит жестко запрограммированные ссылки на другие классы или имена классов;
- внутри класса объекты создаются с использованием оператора `new`;
- у класса есть защищенные свойства или методы, позволяющие изменять его поведение путем переопределения состояния или поведения.

Рефакторинг: абстрактная фабрика

Нам бы хотелось исправить этот неудачный дизайн, который требует от нас постоянного погружения в класс `GenericEncoder` для изменения поведения, ориентированного на формат. Сначала мы должны делегировать ответственность за выбор правильного кодировщика формата другому классу. Когда вы рассматриваете ответственности как причины для изменения (*см.* главу 1), это имеет смысл: логика поиска правильного кодировщика может измениться, поэтому неплохо было бы перенести эту ответственность в другой класс.

Этот новый класс также может быть реализацией шаблона проектирования *абстрактная фабрика*[1]. Абстрактность представле-

[1] *Erich Gamma e. a.* Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

на тем фактом, что метод `create()` обязан возвращать экземпляр данного *интерфейса*. Нас не интересует его фактический класс; мы только хотим получить объект с помощью метода `encode($data)`. Поэтому нам нужен интерфейс для таких кодировщиков. Затем мы удостоверяемся, что каждый существующий кодировщик, ориентированный на конкретный формат, реализует этот интерфейс (*см.* листинг 2-4 и рис. 2-2).

Листинг 2-4. EncoderInterface и его классы реализации

```
/**
* Interface for format-specific encoders
*/
interface EncoderInterface
{
public function encode($data): string;
}
class JsonEncoder implements EncoderInterface
{
// ...
}
class XmlEncoder implements EncoderInterface
{
// ...
}
class YamlEncoder implements EncoderInterface
{
// ...
}
```

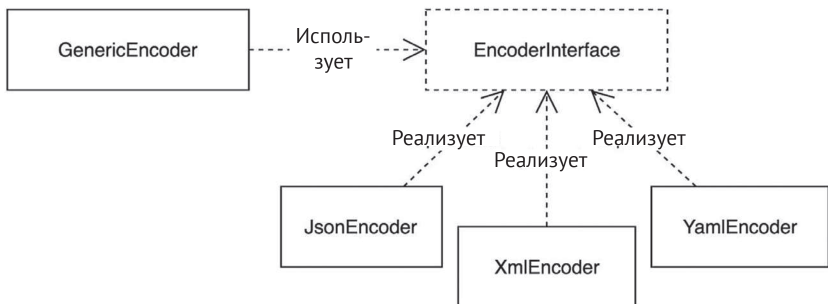


Рис. 2-2. Внедрение интерфейса EncoderInterface

Теперь мы можем переместить логику создания кодировщиков, ориентированных на конкретный формат, в класс именно с этой

ответственностью. Давайте назовем его EncoderFactory (*см.* листинг 2-5).

Листинг 2-5. Класс EncoderFactory

```
class EncoderFactory
{
public function createForFormat(
string $format
) : EncoderInterface {
if ($format === 'json') {
return new JsonEncoder();
} elseif ($format === 'xml') {
return new XmlEncoder();
} elseif (...) {
// ...
}
throw new InvalidArgumentException('Unknown format');
}
}
```

Теперь мы должны убедиться, что класс GenericEncoder больше не создает кодировщики, ориентированные на конкретный формат. Вместо этого он должен делегировать эту задачу классу EncoderFactory, который он получает в качестве аргумента конструктора (*см.* листинг 2-6).

Листинг 2-6. Класс GenericEncoder теперь использует EncoderFactory

```
class GenericEncoder
{
private $encoderFactory;
public function __construct(
EncoderFactory $encoderFactory
) {
$this->encoderFactory = $encoderFactory;
}
public function encodeToFormat($data, string $format): string
{
$encoder = $this->encoderFactory
->createForFormat($format);
$data = $this->prepareData($data, $format);
return $encoder->encode($data);
}
}
```

Оставляя ответственность за создание правильного кодировщика encoderFactory, GenericEncoder теперь соответствует принципу *единственной ответственности*.

Использование фабрики кодировщиков для получения правильного кодировщика заданного формата означает, что добавление дополнительного кодировщика больше *не требует от нас изменения* класса `GenericEncoder`. Вместо этого нам нужно изменить класс `EncoderFactory`.

Но когда мы смотрим на класс `EncoderFactory`, тут все еще находится страшный жестко закодированный список поддерживаемых форматов и соответствующих кодировщиков. Что еще хуже, имена классов по-прежнему жестко закодированы. Это означает, что теперь `EncoderFactory` *закрыт для расширения*.

То есть его поведение нельзя расширить без изменения кода, и тем самым класс нарушает принцип открытости/закрытости.

БЫСТРАЯ ВОЗМОЖНОСТЬ РЕФАКТОРИНГА: ДИНАМИЧЕСКИЕ ИМЕНА КЛАССОВ?

Это кажется легкой добычей. Как вы, возможно, заметили, внутри оператора `switch` существует поразительная симметрия: для формата `json` возвращается экземпляр `JsonEncoder`, для формата `xml` – `XmlEncoder` и т. д. Если ваш язык программирования поддерживает динамические имена классов, как это делает PHP, можно легко преобразовать это в то, что больше не будет жестко закодировано:

```
$class = ucfirst(strtolower($format)) . 'Encoder';
if (!class_exists($class)) {
throw new InvalidArgumentException('Unknown format');
}
```

Да, на самом деле это эквивалентный код. Он короче, устраняет необходимость использования оператора `switch` и даже добавляет немного больше гибкости: чтобы расширить его поведение, вам больше не нужно изменять код. В случае нового кодировщика формата Yaml нам только нужно создать новый класс, который следует соглашению об именовании: `YamlEncoder`. Вот и все.

Тем не менее использование динамических имен классов, чтобы сделать класс расширяемым, как этот, порождает новые проблемы и не устраняет некоторые из существующих проблем:

- использование соглашения об именовании предлагает вам лишь некую гибкость в качестве сопровождающего кода. Когда кому-то нужно добавить поддержку нового формата, он должен поместить класс в ваше пространство имен, что возможно, но не очень удобно для пользователя;

- более серьезная проблема: логика создания по-прежнему ограничена использованием `new ...()`. Например, если в классе кодировщика есть зависимости, их нельзя внедрить (например, в качестве аргументов конструктора). Мы решим эту проблему далее.

Рефакторинг: делаем абстрактную фабрику открытой для расширения

Первое, что мы могли бы сделать, – это применить принцип *инверсии зависимостей* (*см.* главу 5), определив интерфейс для фабрик кодировщиков. `EncoderFactory`, который у нас уже есть, должен реализовывать новый интерфейс, а аргумент конструктора `GenericEncoder` должен иметь интерфейс в качестве своего типа (*см.* листинг 2-7 и рис. 2-3).

Листинг 2-7. Интерфейс для фабрики

```
interface EncoderFactoryInterface
{
public function createForFormat(
string $format
): EncoderInterface;
}
class EncoderFactory implements EncoderFactoryInterface
{
// ...
}
class GenericEncoder
{
public function __construct(
EncoderFactoryInterface $encoderFactory
) {
// ...
}
// ...
}
```

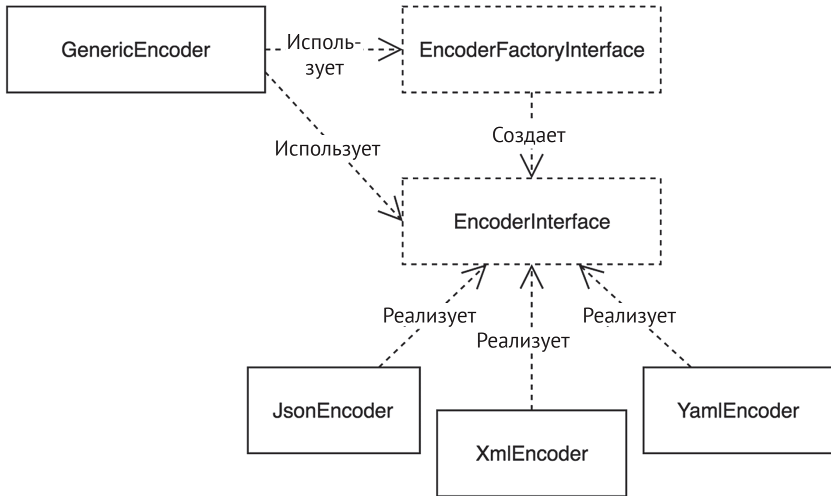


Рис. 2-3. Использование интерфейса EncoderFactoryInterface

Замена или декорирование фабрики кодировщика

Делая класс `GenericEncoder` зависимым от интерфейса, а не от класса, мы добавили к нему первую точку расширения. Пользователям этого класса будет легко полностью заменить фабрику кодировщика. Теперь она представляет собой правильную зависимость, которая вставляется в качестве аргумента конструктора типа `EncoderFactoryInterface` (*см.* листинг 2-8).

Листинг 2-8. Меняем фабрику кодировщика на свою

```
class MyCustomEncoderFactory implements EncoderFactoryInterface
{
// ...
}
$encoder = new GenericEncoder(new MyCustomEncoderFactory());
```

Внедрив интерфейс, мы предоставили пользователям еще одну очень мощную опцию. Возможно, они не хотят полностью заменять существующий `EncoderFactory`, а просто хотят улучшить его поведение. Например, предположим, что они хотят получить кодировщик для заданного формата из локатора служб и вернуться к стандартному `EncoderFactory` в случае, если формат окажется неизвестен. Используя этот интерфейс, они могут *составить* новую

фабрику, которая реализует требуемый интерфейс и получает исходный EncoderFactory в качестве аргумента конструктора (*см.* листинг 2-9). Можно сказать, что новая фабрика «оборачивает» старую. Технически это называется «декорирование».

Листинг 2-9. Декорирование исходного EncoderFactory

```
class MyCustomEncoderFactory implements EncoderFactoryInterface
{
private $fallbackFactory;
private $serviceLocator;
public function __construct(
ServiceLocatorInterface $serviceLocator,
EncoderFactoryInterface $fallbackFactory
) {
$this->serviceLocator = $serviceLocator;
$this->fallbackFactory = $fallbackFactory;
}
public function createForFormat($format): EncoderInterface
{
if ($this->serviceLocator->has($format . '.encoder') {
return $this->serviceLocator
->get($format . '.encoder');
}
return $this->fallbackFactory->createForFormat($format);
}
}
```

Делаем EncoderFactory открытым для расширения

Хорошо, что теперь пользователи могут реализовать свой собственный экземпляр EncoderFactoryInterface или декорировать существующий. Однако принуждение пользователя к повторной реализации EncoderFactoryInterface, просто чтобы добавить поддержку другого формата, кажется несколько неэффективным. Когда появляется новый формат, мы хотим продолжать использовать тот же старый EncoderFactory, но нам нужно поддерживать новый формат, не затрагивая код самого класса. Кроме того, если одному из кодировщиков потребуется другой объект для выполнения своей задачи, предоставить этот объект в качестве аргумента конструктора пока невозможно, поскольку *логика создания* каждого из кодировщиков жестко закодирована в классе EncoderFactory.

Другими словами, расширить или изменить поведение класса EncoderFactory, не модифицируя его, невозможно: логику, с помощью которой фабрика кодировщиков решает, какой кодировщик она

должна создать и как она должна это делать для заданного формата, нельзя изменить извне. Но довольно легко вывести эту логику из класса EncoderFactory, что делает класс *открытым для расширения*.

Есть несколько способов сделать фабрику вроде EncoderFactory открытой для расширения. Я решил добавить в EncoderFactory специальные фабрики, как показано в листинге 2-10.

Листинг 2-10. Внедрение специальных фабрик

```
class EncoderFactory implements EncoderFactoryInterface
{
private $factories = [];
/**
* Регистрируем callable, который возвращает экземпляр
* EncoderInterface для данного формата;
*
* @param string $format
* @param callable $factory
*/
public function addEncoderFactory(
string $format,
callable $factory
): void {
$this->factories[$format] = $factory;
}
public function createForFormat(
string $format
): EncoderInterface {
$factory = $this->factories[$format];
// Фабрика - это псевдотип callable;
$encoder = $factory();
return $encoder;
}
}
```

Для каждого формата можно ввести псевдотип callable[1]. Метод createForFormat () принимает этот псевдотип, вызывает его и использует его возвращаемое значение в качестве фактического кодировщика для данного формата.

Такая полностью динамическая и расширяемая реализация позволяет разработчикам добавлять столько кодировщиков, сколько им захочется. В листинге 2-11 показано, как выглядит внедрение кодировщиков.

---

[1] Также *см.* документацию по типу callable в PHP: https://secure.php.net/manual/ru/language.types.callable.php.

Листинг 2-11. Динамическое определение фабрик кодировщиков

```
$encoderFactory = new EncoderFactory();
$encoderFactory->addEncoderFactory(
'xml',
function () {
return new XmlEncoder();
}
);
$encoderFactory->addEncoderFactory(
'json',
function () {
return new JsonEncoder();
}
);
$genericEncoder = new GenericEncoder($encoderFactory);
$data = ...;
$jsonEncodedData = $genericEncoder->encode($data, 'json');
```

Внедрив фабрики с использованием типа callable, мы освободили `EncoderFactory` от ответственности за предоставление правильных аргументов конструктора для каждого кодировщика. Другими словами, мы поместили знания о *логике создания* за пределами `EncoderFactory`, благодаря чему она одновременно придерживается принципа *единственной ответственности* и *принципа открытости/закрытости*.

ПРЕДПОЧТЕНИЕ НЕИЗМЕНЯЕМЫМ СЕРВИСАМ

Как вы, возможно, заметили, `EncoderFactory` внезапно стала изменяемым сервисом, когда мы добавили метод `addEncoderFactory()`. Это было удобно, но на практике было бы разумно разработать сервис, который будет *неизменяемым*. Чтобы добиться этого, примените следующее правило:

> *После создания экземпляра не должно быть возможности изменить какие-либо свойства сервиса.*

Самым большим преимуществом неизменяемого сервиса является то, что его поведение не изменится при последующих вызовах. Он будет полностью настроен перед первым использованием, и получить каким-либо образом другие результаты при последующих вызовах будет невозможно.

Если вы по-прежнему предпочитаете использовать отдельные методы для настройки объекта, не включайте эти методы в опубли-

кованный интерфейс класса. Они существуют только для клиентов, которым необходимо настроить объект, а не для клиентов, фактически использующих объекты. Например, контейнер внедрения зависимостей вызовет метод `addEncoderFactory()` при настройке нового экземпляра `EncoderFactory`, но обычные клиенты, такие как сам `GenericEncoder`, будут вызывать только метод `createForFormat()`.

Рефакторинг: полиморфизм

Мы приложили усилия для реализации хорошей абстрактной фабрики для кодировщиков, но в классе `GenericEncoder` по-прежнему находится этот уродливый оператор `switch` для подготовки данных перед их кодированием (*см.* листинг 2-12).

Листинг 2-12. Возвращаясь к методу prepareData()

```
class GenericEncoder
{
private function prepareData($data, string $format)
{
switch ($format) {
case 'json':
$data = $this->forceArray($data);
$data = $this->fixKeys($data);
// fall through
case 'xml':
$data = $this->fixAttributes($data);
break;
default:
throw new InvalidArgumentException(
'Format not supported'
);
}
return $data;
}
}
```

Куда мы должны поместить эту логику? Другими словами, на ком лежит *ответственность* за подготовку данных перед их кодированием? Это должен делать класс `GenericEncoder`? Нет, потому что подготовка данных зависит от формата и не является общей. `EncoderFactory`? Нет, потому что он знает только, как создавать кодировщики. Это один из кодировщиков, ориентированных на формат? Да! Они знают все о кодировании данных в собственный формат.

Поэтому давайте делегируем логику «подготовить данные» конкретным кодировщикам, добавив метод prepareData($ data) в EncoderInterface и вызвав его в методе encodeToFormat() класса GenericEncoder.

Листинг 2-13. Перемещение метода prepareData() в EncoderInterface

```
interface EncoderInterface
{
public function encode($data);
/**
* Делаем все необходимое для подготовки данных к кодированию;
*
* @param mixed $data
* @return mixed
*/
public function prepareData($data);
}
class GenericEncoder
{
public function encodeToFormat($data, string $format): string
{
$encoder = $this->encoderFactory
->createForFormat($format);
/*
* За подготовку данных теперь отвечает кодировщик;
*/
$data = $encoder->prepareData($data);
return $encoder->encode($data);
}
}
```

В случае с классом JsonEncoder это будет выглядеть так, как показано в листинге 2-14.

Листинг 2-14. Пример реализации prepareData()

```
class JsonEncoder implements EncoderInterface
{
public function encode($data): string
{
// ...
}
public function prepareData($data)
{
$data = $this->forceArray($data);
$data = $this->fixKeys($data);
return $data;
}
}
```

Это не очень хорошее решение, потому что в нем используется нечто, именуемое «временным зацеплением»: перед вызовом функции `encode()` всегда нужно вызывать функцию `prepareData()`. Если вы этого не сделаете, ваши данные могут быть недействительными и будут не готовы к кодированию.

Поэтому вместо этого нам следует провести *подготовку* части, касающейся данных, *фактического процесса* кодирования внутри кодировщика, предназначенного для конкретного формата. Каждый кодировщик должен сам решить, нужно ли подготовить предоставленные данные перед их кодированием и как это сделать. В листинге 2-15 показано, как это выглядит для кодировщика JSON.

Листинг 2-15. Выполнение подготовки части, касающейся данных в методе encode()

```
class JsonEncoder implements EncoderInterface
{
public function encode($data): string
{
$data = $this->prepareData($data);
return json_encode($data);
}
private function prepareData($data)
{
// ...
return $data;
}
}
```

В этом случае метод `prepareData()` является закрытым. Он не является частью открытого интерфейса кодировщиков, потому что будет применяться только для внутреннего использования. `GenericEncoder` больше не должен вызывать его. Нам нужно только удалить его из `EncoderInterface`, который теперь предоставляет очень чистый API (*см.* листинг 2-16).

Листинг 2-16. EncoderInterface

```
interface EncoderInterface
{
public function encode($data): string;
}
```

Резюмируя, скажем, что класс `GenericEncoder`, с которого мы начали эту главу, был довольно специфичным. Все было жестко

запрограммировано, поэтому было невозможно изменить его поведение, не модифицируя его. Сначала мы перенесли ответственность за создание кодировщиков, специфичных для формата, в фабрику кодировщиков. Затем мы применили немного инверсии зависимостей, введя интерфейс для фабрики кодировщиков. Наконец, мы сделали фабрику кодировщиков полностью динамичной: мы позволили внедрять новые фабрики кодировщиков, зависящие от формата, извне, то есть без изменения кода самой фабрики.

Выполнив все это, мы сделали класс `GenericEncoder` *действительно* общим. Если мы хотим добавить поддержку другого формата, нам больше не нужно изменять его код. Нам нужно только ввести еще один тип callable в фабрику кодировщиков. Это делает оба класса (`GenericEncoder` и `EncoderFactory`) *открытыми для расширения* и *закрытыми для модификации*. Фактически, возможно, больше нет необходимости в классе `GenericEncoder`, если посмотреть, как он выглядит сейчас (*см.* листинг 2-17). Мы могли бы попросить пользователей напрямую вызвать фабрику кодировщиков самостоятельно.

Листинг 2-17. GenericEncoder, возможно, больше не заслуживает того, чтобы быть классом

```
class GenericEncoder
{
public function encodeToFormat($data, string $format): string
{
return $this->encoderFactory
->createForFormat($format)
->encode($data);
}
}
```

Пакеты и принцип открытости/закрытости

Применение принципа *открытости/закрытости* к классам в вашем проекте значительно улучшит реализацию будущих (или измененных) требований для него. Когда поведение класса можно изменить извне, без модификации его кода, люди будут чувствовать себя в безопасности, делая это. Им не нужно будет бояться, что они что-то сломают. Им даже не нужно будет модифицировать существующие модульные тесты для класса.

Когда речь идет о пакетах, принцип *открытости/закрытости* важен по другой причине. Пакет будет использоваться во многих различных проектах и в разных обстоятельствах. Это означает, что классы в пакете не должны быть слишком специфичными и оставлять место для деталей, которые будут реализованы различными способами. И когда поведение *должно* быть специфичным (в какой-то момент пакет должен иметь свое мнение относительно чего-либо), должна быть возможность изменить это поведение без фактического изменения кода. Тем более что большую часть времени пользователи не могут изменить этот код без клонирования и поддержки всего пакета.

Вот почему принцип *открытости/закрытости* крайне полезен и должен широко применяться при разработке классов, которые обязательно окажутся в повторно используемом пакете. На практике это означает, что вы позволяете настраивать классы, внедряя различные аргументы конструктора (что также известно как *внедрение зависимостей*). Для взаимодействующих объектов, которые вы могли извлечь при применении принципа *единственной ответственности*, убедитесь, что у этих объектов есть опубликованный интерфейс, позволяющий пользователям декорировать существующие классы.

Повсеместное применение принципа *открытости/закрытости* позволит изменить поведение любого класса в вашем пакете путем переключения или декорирования только аргументов конструктора. Поскольку пользователям больше не стоит полагаться на подклассы для переопределения поведения класса, это дает вам мощную возможность пометить все из них как `final`[1]. Если вы сделаете это, то лишите пользователей возможности создавать подклассы. Это уменьшает количество возможных вариантов использования, которые необходимо учитывать при внесении изменений в класс. По сути, это поможет вам сохранить обратную совместимость в будущем и даст полную свободу изменять любые детали реализации класса.

Заключение

Обычно, если вы хотите изменить поведение класса, вам нужно изменить его код. Чтобы предотвратить изменение класса, в част-

[1] Отличную дискуссию на тему маркировки классов как final можно найти по адресу: https://ocramius.github.io/blog/when-to-declare-classes-final/.

ности если этот класс является частью пакета, вы должны встроить опции для изменения поведения класса *извне*. Другими словами, у вас должна быть возможность расширить его поведение без изменения какой-либо части его кода.

Вот что значит применять принцип *открытости/закрытости*: мы должны убедиться, что объекты открыты для расширения, но закрыты для модификации. Мы обсудили несколько методов для достижения этой цели:

- применять принцип *единственной ответственности* и извлекать взаимодействующие объекты;
- внедрять взаимодействующие объекты в качестве аргументов конструктора (*внедрение зависимостей*);
- предоставлять *интерфейсы* для взаимодействующих объектов, что позволяет пользователю заменять зависимости или *декорировать* их;
- помечать классы как `final`, чтобы у пользователя не было возможности изменять поведение класса путем расширения.

Глава 3

Принцип подстановки Барбары Лисков

Принцип подстановки Барбары Лисков можно сформулировать так[1]:

> *Производные классы должны быть заменяемы их базовыми классами.*

Забавно, что в названии этого принципа фигурирует фамилия Лисков. Это связано с тем, что принцип был впервые сформулирован (используя другую формулировку) Барбарой Лисков. Но в остальном здесь нет никаких сюрпризов и больших концептуальных скачков. Кажется логичным только то, что наследующие классы, или «подклассы», как их обычно называют, должны заменять свои базовые, или «родительские», классы. Но, конечно же, это еще не все. Данный принцип не просто утверждение очевидного.

Анализируя его, мы признаем две концептуальные части. Сначала речь идет о производных и базовых классах. Потом речь идет о замене.

Хорошо то, что из опыта мы уже знаем, что такое *производный класс*: это класс, который расширяет какой-либо другой класс: *базовый*. В зависимости от языка программирования, с которым вы работаете, базовый класс может быть конкретным классом, абстрактным классом или интерфейсом. Если базовый класс является *конкретным*, у него нет «отсутствующих» (также известных как *виртуальных*) методов. В этом случае производный класс или подкласс переопределяет один или несколько методов, которые уже реализо-

---

[1] *Robert C. Martin*. Principles of OOD // http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

ваны в родительском классе. С другой стороны, если базовый класс является *абстрактным*, существует один или несколько *чистых виртуальных методов*, которые должны быть реализованы производным классом. Наконец, если *все* методы базового класса являются чисто виртуальными методами (то есть у них есть только сигнатура и нет тела), то обычно базовый класс называется *интерфейсом*.

Чтобы убедиться, что мы не запутались, взгляните на листинг 3-1, где дается объяснение терминов *базовый класс*, *производный класс* и *интерфейс* с использованием кода на PHP.

Листинг 3-1. Различия между интерфейсом, абстрактным классом и конкретным классом

```
/**
* Конкретный класс: все методы реализованы, но они могут быть
переопределены производными * классами;
*/
class ConcreteClass
{
public function implementedMethod()
{
// ...
}
}
/**
* Абстрактный класс: некоторые методы должны быть реализованы производными
классами;
*/
abstract class AbstractClass
{
abstract public function abstractMethod();
public function implementedMethod()
{
// ...
}
}
/**
* Интерфейс: все методы должны быть реализованы производными классами;
*/
interface AnInterface
{
public function abstractMethod();
}
```

Теперь мы знаем все о базовых и производных классах. Но что означает, что производные классы могут быть *заменяемыми*? Кажется, тут есть много возможностей для обсуждения. В целом быть

заменяемым – значит *вести себя хорошо* как подкласс или класс, реализуя интерфейс. «Вести себя хорошо» значит вести себя «как ожидалось» или «как было согласовано».

Объединяя обе концепции, принцип *подстановки Барбары Лисков* гласит, что если мы создаем класс, который расширяет другой класс или реализует интерфейс, он должен вести себя так, как и ожидалось.

Такие слова, как «вести себя, как и ожидалось», по-прежнему довольно расплывчаты. Вот почему указывать на нарушения принципа *подстановки Лисков* может быть довольно сложно. Среди разработчиков даже могут возникать разногласия по поводу того, что считать нарушением этого принципа. Иногда это дело вкуса. А иногда это зависит от самого языка программирования и конструкций объектно-ориентированного программирования, которые он предлагает.

Тем не менее мы можем указать на некоторые общие неудачные практики, которые могут помешать классам быть хорошей заменой своих родительских классов или хорошей реализацией интерфейса. Поэтому, хотя сам принцип изложен в позитивном ключе, ниже следует обсуждение некоторых повторяющихся нарушений этого принципа. Это даст вам представление о том, что означает вести себя *плохо* в качестве замены класса или интерфейса, и косвенно поможет вам сформировать представление о том, как вести себя *хорошо,* будучи производным классом.

Нарушение: производный класс не имеет реализации всех методов

Когда класс не имеет надлежащей реализации всех методов своего родительского класса (или его интерфейса), это приводит к явному нарушению принципа *подстановки Лисков*. Это плохое поведение заменителей, когда они не делают все то, что *должны* делать. Рассмотрим, например, этот интерфейс `FileInterface` (*см.* листинг 3-2).

Листинг 3-2. FileInterface

```
interface FileInterface
{
public function rename(string $name): void;
public function changeOwner(string $user, string $group): void;
}
```

Может показаться очевидным, что у файла всегда есть имя и владелец, и их можно изменить. Но также можно себе представить, что для некоторых файлов смена владельца будет вообще невозможна. Возьмем, к примеру, файлы, которые хранятся в хранилище, предоставляемом облачным провайдером, таким как Dropbox. Если мы создаем реализацию `FileInterface` для Dropbox, то должны запретить пользователям пытаться изменить владельца файла, потому что в случае с файлом Dropbox это просто не сработает (*см.* листинг 3-3).

Листинг 3-3. Реализация FileInterface для Dropbox

```
class DropboxFile implements FileInterface
{
public function rename(string $name): void
{
// ...
}
public function changeOwner(string $user, string $group): void
{
throw new BadMethodCallException(
'Not implemented for Dropbox files'
);
}
}
```

Выброс исключений, подобных этому, следует считать плохим поведением для замены: когда кто-то вызывает метод `ChangeOwner()` из класса `DropboxFile`, все приложение может аварийно завершить работу без какого-либо предупреждения.

Фактически мы не хотим, чтобы пользователь вызывал `FileInterface::changeOwner()`, если класс, который он использует, – это `DropboxFile`. Поэтому, возможно, мы можем попросить пользователя выполнить проверку, используя простой оператор условия (*см.* листинг 3-4).

Листинг 3-4. Использование оператора условия для проверки возможности изменения владельца файла

```
if (!$file instanceof DropboxFile) {
$file->changeOwner(...);
}
```

Это предотвратит появление неприятного исключения внутри `changeOwner()`. К сожалению, данное решение не жизнеспособное. Скорее всего, эти строки будут повторяться на протяжении всей

кодовой базы пользователя, что быстро станет для него бременем при обслуживании.

Вместо того чтобы выбрасывать исключение, можно просто скрыть тот факт, что вы не можете изменить владельца DropboxFile (*см.* листинг 3-5).

Листинг 3-5. Молча пропускаем нереализованные методы

```
class DropboxFile implements FileInterface
{
// ...
public function changeOwner(string $user, string $group): void
{
// тсс... это не поддерживается, но зачем кому-то знать об этом?
}
}
```

Реализация данного простого решения может быть очень заманчивой. К сожалению, мы не можем этого сделать – смена владельца файла является *важной операцией*. В конце концов, речь идет о безопасности. Некоторые другие части системы могут рассчитывать на DropboxFile::changeOwner (), чтобы *действительно* изменить владельца файла; например, чтобы сделать его недоступным для предыдущего владельца. Если по какой-либо причине смена владельца данного типа файла невозможна, это должно быть оговорено в контракте. Другими словами, его интерфейс не должен предлагать метод, который делает так, что это *кажется* возможным.

Наиболее подходящим решением было бы разделить интерфейс (*см.* листинг 3-6).

Листинг 3-6. Разделение FileInterface

```
interface FileInterface
{
public function rename(string $name): void;
}
interface FileWithOwnerInterface extends FileInterface
{
public function changeOwner(
string $user,
string $group
): void;
}
```

Вместе эти интерфейсы образуют иерархию файловых типов. Сначала идет универсальный файловый тип, определенный `FileInterface` (который предлагает только метод для его переименования). Затем идет подтип файлов, владельца которого можно поменять. После того как мы определили эти интерфейсы, класс `DropboxFile` будет реализовывать только общий интерфейс `FileInterface` (*см.* листинг 3-7).

Листинг 3-7. DropboxFile больше не нужно реализовывать метод changeOwner()

```
class DropboxFile implements FileInterface
{
public function rename($name)
{
// ...
}
}
```

И любой другой файловый тип, который поддерживает смену владельца, например `LocalFile`, реализует `FileWithOwnerInterface` (*см.* листинг 3-8).

Листинг 3-8. Класс LocalFile реализует методы rename() и changeOwner()

```
class LocalFile implements FileWithOwnerInterface
{
public function rename(string $name): void
{
// ...
}
public function changeOwner(string $user, string $group): void
{
// ...
}
}
```

На рис. 3-1 показана получившаяся иерархия классов.

Наконец, мы снова заставили код придерживаться принципа *подстановки Барбары Лисков*. Все производные классы (`DropboxFile` и `LocalFile`) теперь ведут себя хорошо в качестве замены своих базовых классов (`FileInterface` и `FileWithOwnerInterface`), а все методы базовых классов правильно реализованы в производных классах.

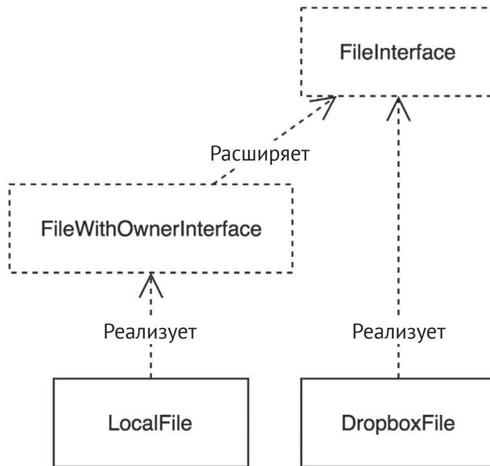


Рис. 3-1. Новая иерархия файловых классов

Протекающие абстракции

Когда мы начинали с этого примера, `FileInterface` задумывался как абстракция для всех файлов. Идея `FileInterface` заключалась в том, чтобы представить некие действия («переименовать» или «изменить владельца»), которые подойдут для любого файла, независимо от того, где он хранится.

Пытаясь абстрагировать (то есть убрать детали) конкретные файлы и их поведение, мы сделали неверное предположение, согласно которому *любая* реализация `FileInterface` сможет обеспечить значимые реализации *всех* методов этого интерфейса.

Однако когда мы приступили к реализации `FileInterface` для файла, хранящегося в Dropbox, это предположение оказалось неверным. Оказалось, что `FileInterface` – это неправильное обобщение понятия «файл». Такое обобщение обычно называют *протекающей абстракцией*. Этот термин стал известен благодаря Джоэлю Спольски[1], когда он сформулировал свой *закон протекающих абстракций*:

Любые нетривиальные абстракции в какой-то степени подвержены утечкам.

---

1 *Joel Spolsky*. The Law of Leaky Abstractions // https://www.joelonsoftware.com/2002/11/11/the-law-of-leaky-abstractions/.

Будучи программистами, мы все время ищем абстракции. Мы хотим рассматривать нечто конкретное как более общее. Когда мы делаем это последовательно, то можем впоследствии безбоязненно заменить любую конкретную вещь какой-либо другой конкретной вещью. Система не сломается, потому что каждая ее часть зависит только от абстрактных вещей и не заботится о специфике.

Проблема с большинством (всеми?) абстракций, как гласит *закон протекающих абстракций*, заключается в том, что они подвержены утечкам, а это значит, что вам никогда не удастся абстрагироваться от базовой специфики. Некоторые базовые детали обязательно появятся и встанут у нас на пути.

Однако пока вы знаете об этом ограничении, вы все равно сможете разрабатывать абстрактные вещи, которые являются абстрактными только до определенного момента. Просто убедитесь, что абстракция соответствует вашим целям, и не пытайтесь вместить все возможные специфичные вещи, какие есть, в свою абстракцию. Среди ученых (и приверженцев предметно-ориентированного проектирования) этот совет известен как:

По сути, все модели ошибочны, но некоторые из них полезны[1].

Нарушение: разные замены возвращают вещи разных типов

Это нарушение касается, в частности, языков программирования, которые не являются строго типизированными, например PHP. Эти языки допускают большую неопределенность в отношении типа, например, возвращаемых значений.

Если в языке программирования нет возможности определить тип возвращаемого значения метода, можно использовать распространенное решение, когда тип упоминается в блоке метода (*см.* листинг 3-9).

Листинг 3-9. Интерфейс RouterInterface

```
interface RouterInterface
{
/**
* @return Route[]
```

[1] *Box G. E. P., Draper N. R.* (1987) Empirical Model-Building and Response Surfaces. John Wiley & Sons.

```
*/
public function getRoutes();
// ...
}
```

Метод `getRoutes()` должен возвращать что-то итерируемое (отсюда и знак []), содержащее объекты `Route`. Но разные реализации маршрутизатора могут возвращать разные типы итерируемых вещей, допустим массивы или (в PHP) объект, который реализует тип `Traversable`[1]. Например, класс `SimpleRouter` возвращает простой массив объектов `Route` (*см.* листинг 3-10).

Листинг 3-10. Реализация метода getRoutes(), который возвращает массив

```
class SimpleRouter implements RouterInterface
{
public function getRoutes()
{
$routes = [];
// Добавляем объекты Route в $routes;
$routes[] = ...;
return $routes;
}
}
```

Но класс `AdvancedRouter` возвращает гораздо более продвинутый объект `RouteCollection` (*см.* листинг 3-11), который реализует тип `Traversable` (фактически он реализует `Iterator`[2], который, в свою очередь, реализует `Traversable`).

Листинг 3-11. Реализация метода getRoutes(), который возвращает RouteCollection

```
class AdvancedRouter implements RouterInterface
{
public function getRoutes()
{
$routeCollection = new RouteCollection();
// ...
return $routeCollection;
}
}
```

---

[1] *См.* документацию по типу `Traversable` на странице: https://secure.php.net/traversable.

[2] *См.* документацию по типу `Iterator` на странице: https://secure.php.net/iterator.

```
class RouteCollection implements Iterator
{
// ...
}
```

Теперь классы AdvancedRouter и SimpleRouter выглядят хорошими заменителями интерфейса RouterInterface, но на самом деле они таковыми не являются. Хотя оба класса и реализуют метод getRoutes(), они возвращают значение другого типа.

Это нарушение принципа *подстановки Барбары Лисков* может оставаться незамеченным некоторое время, когда итерируется только возвращаемое значение getRoutes(), используя простой цикл foreach (*см.* листинг 3-12).

Листинг 3-12. Итерация возвращаемого значения getRoutes()

```
// $router реализует RouterInterface, поэтому $routes является итерируемым;
$routes = $router->getRoutes();
foreach ($routes as $route) {
// $route - объект Route;
}
```

Это должно работать во всех ситуациях, потому что цикл foreach перебирает значения в массиве, а также значения, предоставляемые итератором. Но так как многие вещи, которые являются итерируемыми (или, по крайней мере, все массивы), также являются вычисляемыми, однажды кто-то может попытаться использовать функцию count() для возвращаемого значения getRoutes() (*см.* листинг 3-13).

Листинг 3-13. Подсчет возвращаемого значения getRoutes()

```
if (count($routes) > 10) {
// ...
}
```

При использовании класса SimpleRouter это будет работать, а при использовании класса AdvancedRouter нет, поскольку класс RouteCollection не реализует тип Countable[1]. Таким образом, становится ясно, что существует проблема с взаимосвязью между родительскими и производными классами.

В отличие от предыдущего нарушения, которое мы обсуждали, проблема состоит не в том, что классы SimpleRouter и AdvancedRouter

---

[1] *См*. документацию по типу Countable по адресу: https://secure.php.net/countable.

являются плохой заменой RouterInterface. Реальная проблема заключается в неоднозначно определяемом типе возвращаемого значения метода getRoutes():Route[].

Решение проблемы состоит в том, чтобы более точно определить тип возвращаемого значения и не допустить случайных отклонений от ожидаемого типа. Поэтому интерфейсы и абстрактные классы всегда должны строго документировать свои типы возвращаемых значений, используя конкретные типы[1] (в листинге 3-14 показан пример этого).

Листинг 3-14. Документирование типов возвращаемых значений

```
/**
* @return array<Route>
*/
```

Однако когда мы используем тип array, все равно остаются вопросы. Массивы представляют собой довольно расплывчатые структуры данных. Разработчик этого интерфейса может задаться вопросом, какой тип ключей ему следует использовать: целые числа или строки. И каковы их ожидаемые значения?

Поскольку эта двусмысленность еще присутствует, желательно ввести новый тип, чтобы не было никаких сомнений. Например, можно было бы определить интерфейс RouteCollectionInterface, чтобы предоставить контракт для типа возвращаемого значения метода getRoutes() (*см.* листинг 3-15).

Листинг 3-15. Интерфейс RouteCollectionInterface

```
interface RouterInterface
{
public function getRoutes(): RouteCollectionInterface;
}
interface RouteCollectionInterface extends Iterator, Countable
{
}
```

С введением этого нового интерфейса для коллекций маршрутов производным классам (т. е. SimpleRouter и AdvancedRouter) будет намного проще вести себя надлежащим образом в качестве замены их базового класса (т. е. RouterInterface). Теперь понятно, что должен возвращать их метод getRoutes().

---

[1] *См.* неофициальное руководство по документированию типов в коде PHP, предложенное phpDocumentor, на странице: https://docs.phpdoc.org/references/phpdoc/types.html.

На рис. 3-2 показано, как выглядит иерархия классов после последних изменений.

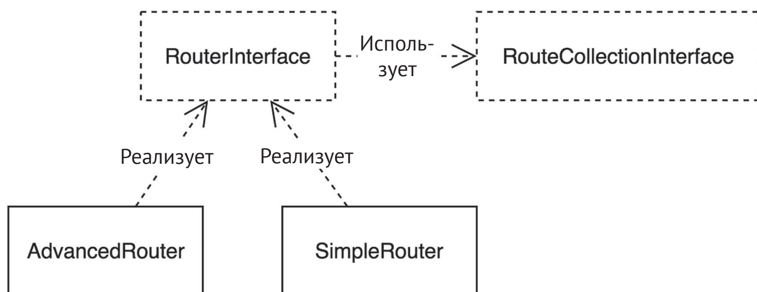


Рис. 3-2. Новая диаграмма зависимостей

Допустимы более конкретные типы возвращаемых значений

Принцип *подстановки Барбары Лисков* не допускает неправильных или неопределенных типов возвращаемых значений. Тем не менее наследующие классы могут возвращать значения, которые являются *подтипом* типа, предписанного базовым классом.

Рассмотрим тип `RouteCollectionInterface` – любого значения, являющегося объектом, реализующим этот интерфейс, будет достаточно в качестве правильного возвращаемого значения, например экземпляра `SomeSpecificRouteCollectionClass` (*см.* листинг 3-16).

Листинг 3-16. SomeSpecificRouteCollectionClass

```
class SomeSpecificRouteCollectionClass
implements RouteCollectionInterface
{
// ...
}
```

Любой класс route collection является производным классом `RouteCollectionInterface`, и, следовательно, он разрешен в качестве возвращаемого значения. Но то же самое относится к любому классу, который расширяет такой класс, потому что расширяемый класс также должен быть хорошей заменой `RouteCollectionInterface`.

Нарушение: производный класс менее снисходителен касательно аргументов метода

Как мы видели ранее, быть хорошей заменой означает реализовать все необходимые методы и заставить их возвращать правильные вещи в соответствии с контрактом базового класса. Когда дело доходит до аргументов метода, замена должна быть *такой же или более снисходительной*, чем это определяет контракт.

Метод должен быть «более или менее снисходительным» относительно аргументов метода – что это означает? Что же, давайте посмотрим на так называемый «почтовый клиент для массовой рассылки». Его интерфейс говорит, что у него должен быть единственный метод: `sendMail()` (*см.* листинг 3-17).

Листинг 3-17. Интерфейс MassMailerInterface

```
interface MassMailerInterface
{
public function sendMail(
TransportInterface $ransport,
Message $message,
Recipients $recipients
): void;
}
```

Производные классы этого интерфейса (то есть базовые классы) должны использовать предоставленный транспорт почты для отправки сообщения всем получателям одновременно. `TransportInterface` скрывает грязные подробности о том, как физически отправлять сообщение получателям. Например, тут могут быть реализации `TransportInterface`, которые используют `sendmail`, протокол SMTP или встроенную PHP-функцию `mail()` для доставки почты.

В листинге 3-18 приводится частичная реализация `MassMailerInterface`, где используется протокол SMTP для отправки электронной почты множеству получателей одновременно. Первое, что он делает, – это проверяет, что пользователь предоставил правильный тип аргумента для `$transport` (в конце концов, этот класс работает только с SMTP).

Листинг 3-18. Класс SmtpMassMailer реализует MassMailerInterface

```
class SmtpMassMailer implements MassMailerInterface
{
```

```
public function sendMail(
TransportInterface $transport,
Message $message,
Recipients $recipients
): void {
if (!($transport instanceof SmtpTransport)) {
throw new InvalidArgumentException(
'SmtpMassMailer only works with SMTP'
);
}
// ...
}
}
```

Ограничивая набор разрешенных аргументов – от всех экземпляров `TransportInterface` только до экземпляров `SmtpTransport`, – `SmtpMassMailer` нарушает принцип *подстановки Барбары Лисков*. В качестве замены базового класса `MassMailerInterface` он должен работать с *любым* транспортом почты, пока он является объектом типа `TransportInterface`. Вместо этого класс `SmtpMassMailer` *менее снисходителен* в отношении аргументов метода, по сравнению с базовым классом. Это *плохое поведение*.

Единственный способ исправить ситуацию – убедиться, что контракт базового класса лучше отражает потребности производных классов. По-видимому, `TransportInterface` как тип для `$transport` недостаточно специфичен, поскольку оказывается, что не каждый вид почтового протокола подходит для массовой рассылки.

Всякий раз, когда мы рассуждаем о проектировании классов, как этот, мы должны следить за такими фразами:

> *не каждый ... является ...*
> *не каждый ... может быть использован как ...*

Обычно они указывают на то, что с иерархией типов наших классов что-то не так. В этой конкретной ситуации наши базовые классы / интерфейсы должны отражать, что существуют различные виды технологии, используемой для передачи сообщения клиенту (mail transport). Переопределив нашу иерархию классов, мы могли бы определить общий интерфейс `TransportInterface` и один специальный интерфейс `TransportWithMassMailSupportInterface`, который расширяет `TransportInterface`.

`SmtpTransport` должен затем реализовать `TransportWithMassMailSupportInterface`, а другие протоколы просто реализуют `TransportInterface` (*см.* листинг 3-19 и рис. 3-3).

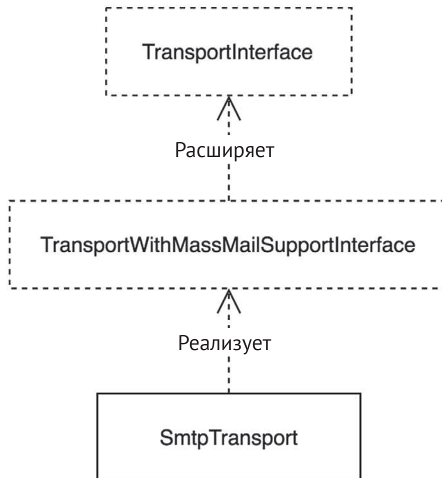


Рис. 3-3. Новая диаграмма классов

Листинг 3-19. Реализация нового интерфейса, поддерживающего массовую рассылку

```
class SmtpTransport implements
TransportWithMassMailSupportInterface
{
// ...
}
```

Наконец, мы можем изменить ожидаемый тип аргумента `$transport` на `TransportWithMassMailSupportInterface`, чтобы предотвратить предоставление неверного типа транспорта методу `sendMail()` (*см.* листинг 3-20).

Листинг 3-20. Использование более специфичного типа для параметра transport

```
interface MassMailerInterface
{
public function sendMail(
TransportWithMassMailSupportInterface $transport,
Message $message,
Recipients $recipients
);
}
```

Затем мы можем изменить `SmtpMassMailer` и удалить дополнительную проверку типа предоставленного аргумента `$transport` (*см.* листинг 3-21).

Листинг 3-21. Больше не нужно ограничивать использование метода sendMail()

```
class SmtpMassMailer implements MassMailerInterface
{
public function sendMail(
TransportWithMassMailSupportInterface $transport,
Message $message,
Recipients $recipients
): void {
/*
* Больше не нужно проверять $transport, он поддерживает массовую рассылку;
*/
// ...
}
}
```

Наконец, класс `SmtpMassMailer` придерживается принципа *подстановки Барбары Лисков*. Он хорошо ведет себя в качестве замены, потому что не накладывает дополнительных ограничений на входные аргументы, как это делает его базовый класс (`MassMailerInterface`).

В зависимости от вашей конкретной ситуации вводить этот дополнительный уровень абстракции может быть неоправданно. Может быть, вы пытаетесь переопределить вещи абстрактно, но на самом деле это просто *конкретные вещи*. Или, может быть, вы пытаетесь найти сходства, которые нельзя найти, потому что их не существует. Например, может быть невозможно использовать какой-либо другой транспорт для массовой рассылки, кроме SMTP. Это означает, что не может быть другого средства для массовой рассылки почты, кроме `SmtpMassMailer`. Тогда мы могли бы также пропустить неуместную абстракцию под названием `MassMailerInterface` и определить один конкретный класс, который подходит для массовой рассылки (*см.* листинг 3-22).

Листинг 3-22. Удаление абстракции – тоже вариант

```
class SmtpMassMailer
{
public function sendMail(
SmtpTransport $transport,
```

```
Message $message,
Recipients $recipients
): void {
// ...
}
}
```

Поскольку класс `SmtpMassMailer` не является производным от базового класса, он больше не нарушает принцип *подстановки Лисков*.

Нарушение: тайное программирование более специфического типа

Базовые классы, такие как интерфейсы, используются для предоставления доступа к явному общедоступному API. Например, общедоступный API-интерфейс `HttpKernelInterface` состоит только из одного метода, который по определению является открытым (*см.* листинг 3-23).

Листинг 3-23. Интерфейс HttpKernelInterface

```
interface HttpKernelInterface
{
public function handle(Request $request): Response;
}
```

Иногда производные классы имеют дополнительные открытые методы. Эти методы составляют его *неявный* общедоступный API. `getEnvironment()` является примером такого метода (*см.* листинг 3-24).

Листинг 3-24. Класс HttpKernel добавляет еще один открытый метод: getEnvironment()

```
class HttpKernel implements HttpKernelInterface
{
public function handle(Request $request): Response
{
// ...
}
public function getEnvironment(): string
{
// ...
}
}
```

Метод getEnvironment() не определен в HttpKernelInterface. Поэтому, когда вы хотите использовать этот метод, вы должны явно зависеть от класса HttpKernel, а не от интерфейса, как это делает CachedHttpKernel (*см.* листинг 3-25). Он оборачивает экземпляр HttpKernel и добавляет дополнительный функционал HTTP-кеширования.

Листинг 3-25. Реализация HttpKernelInterface: CachedHttpKernel

```
class CachedHttpKernel implements HttpKernelInterface
{
public function __construct(HttpKernel $kernel)
{
if ($kernel->getEnvironment() === 'dev') {
// ...
}
}
public function handle(Request $request): Response
{
// ...
}
}
```

Будучи создателями класса CachedHttpKernel, у нас может возникнуть желание сделать его немного более универсальным, позволив пользователям оборачивать *любой* экземпляр HttpKernelInterface. Для этого нужна всего лишь простая модификация конструктора класса, как показано в листинге 3-26.

Листинг 3-26. Вызов метода getEnvironment() для экземпляра HttpKernelInterface

```
class CachedHttpKernel implements HttpKernelInterface
{
public function __construct(HttpKernelInterface $kernel)
{
if ($kernel->getEnvironment() === 'dev') {
// ...
}
}
// ...
}
```

Возможно, вы уже заметили проблему: мы по-прежнему используем метод getEnvironment(), который был допустим, когда $kernel гарантированно был экземпляром HttpKernel. После изменения типа аргумента конструктора мы больше не можем быть уверены

в этом. Теперь мы знаем только, что `$kernel` является экземпляром `HttpKernelInterface`.

Предоставленный аргумент по-прежнему может быть экземпляром `HttpKernel`, и поскольку данный пример написан на PHP, в этом случае код будет работать отлично. Валидность кода определяется только во время выполнения, поэтому даже если типы не совпадают, мы все равно могли бы вызвать метод `getEnvironment()` для `$kernel`.

Таким образом, `CachedKernel` притворяется, что является частью прекрасной иерархии замещаемых классов, хотя на самом деле это не так. Это нарушает традицию реализации и требует только метода `handle()` интерфейса `KernelInterface`, тем самым нарушая принцип *подстановки Барбары Лисков*.

Решение данной проблемы состоит в том, чтобы быть более осторожным в отношении соблюдения контрактов базового класса. Например, можно было бы расширить интерфейс, чтобы он содержал необходимый метод `getEnvironment()` (*см.* листинг 3-27).

Листинг 3-27. Добавляем метод getEnvironment() в KernelInterface

```
interface KernelInterface
{
public function handle(Request $request): Response;
public function getEnvironment(): string;
}
```

Или можно было бы разделить интерфейс, как мы делали это в нескольких предыдущих случаях, поэтому вам могут потребоваться более специфичные типы объектов (*см.* листинг 3-28).

Листинг 3-28. Разделение интерфейса

```
interface HttpKernelInterface
{
public function handle(Request $request): Response;
}
interface HttpKernelWithEnvInterface
extends HttpKernelInterface
{
public function getEnvironment(): string;
}
class CachedHttpKernel implements HttpKernelInterface
{
public function __construct(
HttpKernelWithEnvironmentInterface $kernel
) {
```

```
// ...
}
}
```

В качестве последнего средства можно проверить, что фактический аргумент реализует желаемый интерфейс, прежде чем вызывать его метод `getEnvironment()` (как показано в листинге 3-29).

Листинг 3-29. Проверка на наличие специфичного типа

```
class CachedHttpKernel implements HttpKernelInterface
{
public function __construct(HttpKernelInterface $kernel)
{
if ($kernel instanceof HttpKernelWithEnvInterface) {
$environment = $kernel->getEnvironment();
// ...
}
}
}
```

Помимо того что это специальное решение очень похоже на то, что здесь, существует упущенная возможность для полиморфизма. Мы хотели бы всегда иметь возможность вызывать метод `getEnvironment()` для экземпляра `HttpKernelInterface`. Так почему бы не добавить его в интерфейс, убедившись, что все реализации вернут правильное значение при вызове этого метода?

Пакеты и принцип подстановки Барбары Лисков

Принцип *подстановки Барбары Лисков* важен для вас как разработчика пакетов с двух сторон: во-первых, когда ваш пакет определяет интерфейс (или базовый класс) и, во-вторых, когда ваш пакет обеспечивает реализацию некоего интерфейса (или базового класса), потенциально интерфейса из другого пакета.

Определение интерфейса, как мы узнали в предыдущей главе, полезно, когда вы хотите предоставить пользователю точку расширения с помощью *внедрения зависимостей* (и, возможно, *декорирования*). Если вы предоставляете новый интерфейс, убедитесь, что по крайней мере есть смысл для пользователя обеспечить альтернативные реализации, которые по-прежнему могут быть правильной заменой этому интерфейсу. Следите за своими абстракциями, чтобы все было правильно, и не заставляйте пользователей прибегать к реализации методов, которые не имеют

смысла в *их* контексте. Иногда это означает, что нужно сделать более узкий интерфейс, предложить несколько интерфейсов, которые можно реализовать по отдельности, или переключиться на другой уровень абстракции. Все эти случаи будут рассмотрены более подробно в главе 4, когда мы будем обсуждать принцип *разделения интерфейса*.

Вы можете упростить разработчикам реализацию подходящей замены интерфейса, который вы внедряете, указав типы аргументов и возвращаемые значения. Как мы видели в примере с методом `getRoutes()`, возвращающим объект `RouteCollectionInterface`, ситуацию можно радикально улучшить путем введения выделенных типов. Это особенно верно, когда встроенные типы языка программирования не являются полезными или недостаточно специфичны, или если он даже не поддерживает правильную типизацию на всех уровнях.

Если ваш пакет содержит класс, который реализует некий интерфейс, убедитесь, что этот класс придерживается контракта, переданного этим интерфейсом, и сопровождающей его документации. Другими словами, убедитесь, что ваша реализация является хорошей заменой для данного интерфейса («базовый класс»). Таким образом, пользователь не запутается, когда будет переходить с одной реализации на другую и что-то внезапно перестанет работать или будет работать немного иначе.

Наконец, всегда рассматривайте возможность *не предоставлять пользователю интерфейс* или базовый класс для реализации либо наследования. Это может снизить гибкость или расширяемость вашего пакета, но также приведет к тому, что ваш пакет будет «сам себе на уме». Это часто имеет эффект, при котором облегчается понимание и работа с пакетом. Некоторые вещи не должны заменяться пользователями; они просто такие, какими вы хотите, чтобы они были, или такие, как вам кажется, они имеют наибольшее значение. Может случиться так, что часть, которую пользователь не может перенастроить, заменить или переопределить, также является частью, которая выделяет ваш пакет среди более общих пакетов.

Заключение

Принцип *подстановки Барбары Лисков* требует от производных классов, чтобы они были хорошей заменой. Мы обсудили несколько

примеров плохой замены. Основываясь на этих негативных примерах, мы можем составить представление о том, что значит «быть хорошей заменой». Хорошая замена:

- обеспечивает реализацию для всех методов базового класса;
- возвращает тип вещей, которые предписывает базовый класс (или более специфичные типы);
- не накладывает дополнительных ограничений на аргументы методов;
- не использует нестрогую типизацию для разрыва контракта, который был предоставлен базовым классом.

Глава 4

Принцип разделения интерфейса

Четвертый принцип проектирования классов в ООП – это принцип *разделения интерфейса*. Вот как он звучит[1]:

Создавайте узкоспециализированные интерфейсы, предназначенные для конкретного клиента.

«Узкоспециализированные интерфейсы» означают интерфейсы с небольшим количеством методов. «Предназначенные для конкретного клиента» означает, что интерфейсы должны определять методы, которые имеют смысл с точки зрения клиента, *использующего* интерфейс.

Чтобы достичь понимания этого принципа, мы, как и в предыдущей главе, обсудим некоторые его распространенные нарушения. Каждое нарушение сопровождается изменением кода, который решит проблему.

Нарушение: многократные варианты использования

Иногда интерфейс класса (т. е. его общедоступный API) содержит слишком много методов, потому что он обслуживает несколько вариантов использования. Некоторые клиенты объекта будут вызывать другой набор методов, который отличается от того, что вызывают иные клиенты того же объекта.

Почти каждая существующая реализация *контейнера служб* служит отличным примером класса, у которого разные клиенты, поскольку многие контейнеры служб используются как в качестве контейнера внедрения зависимостей (или инверсии управления), так *и* в качестве локатора служб.

Контейнер служб – это объект, который вы используете для извлечения других объектов (т. е. сервисов), как показано в листинге 4-1.

---

[1] *Robert C. Martin.* The Principles of OOD // http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

Листинг 4-1. ServiceContainerInterface

```
interface ServiceContainerInterface
{
public function get(string $name);
// ...
}
// $serviceContainer - это экземпляр ServiceContainerInterface;
$mailer = $serviceContainer->get('mailer');
```

Служба `mailer` вернет полностью инициализированный объект, который можно использовать в качестве почтового клиента. Это разрешает отложенную загрузку сервисов. Поскольку контейнер служб находит службы за вас, его также называют «локатором служб» (подробнее о том, почему использование локатора служб в большинстве случаев не рекомендуется, читайте в этой статье Пола М. Джонса[1]).

Прежде чем вы сможете получить службу из контейнера, какая-то другая часть системы должна правильно его настроить. Контейнер должен быть проинструктирован, как *инициализировать* такие службы, как `mailer`. Это аспект контейнера, который делает его контейнером *внедрения зависимостей*. Посмотрите, как это делается, в листинге 4-2.

Листинг 4-2. Настройка контейнера служб с помощью метода set()

```
interface ServiceContainerInterface
{
public function get(string $name);
public function set(string $name, callable $factory): void;
}
// $serviceContainer - это экземпляр ServiceContainerInterface;
// Настраиваем службу mailer;
$serviceContainer->set(
'mailer',
function () use ($serviceContainer) {
return new Mailer(
// a mailer needs a transport
$serviceContainer->get('mailer.transport')
);
}
);
// Настраиваем службу mailer transport;
```

---

[1] *Paul M. Jones.* Quicker, Easier, More Seductive: Restraining Your Service Locators // http://paul-m-jones.com/archives/4792.

```
$serviceContainer->set(
'mailer.transport',
function () use ($serviceContainer) {
return new MailerSmtpTransport();
}
);
```

Другим частям приложения больше не нужно беспокоиться о том, как инстанцировать и инициализировать службу `mailer` – вся логика создания обрабатывается чем-то другим – контейнером внедрения зависимостей. Вот почему такой контейнер часто называют *контейнером инверсии управления* (*IoC*).

Интересно то, что сценарий использования *настройки* контейнера служб (то есть предоставления ему инструкций по поводу того, как создавать экземпляры служб) полностью отличается от варианта использования при извлечении служб из контейнера (то есть использования его в качестве локатора служб). Тем не менее оба варианта использования будут предоставлены одним и тем же контейнером служб, поскольку оба метода – `get()` и `set()` – определены в `ServiceContainerInterface`.

Это означает, что любой клиент, который зависит от `ServiceContainerInterface`, может как *извлекать* ранее определенные службы, так и определять новые. В действительности большинство клиентов `ServiceContainerInterface` выполняют только одну из этих задач. Клиент либо настраивает контейнер служб (например, когда приложение загружается), *либо* извлекает из него службу (когда приложение запущено и работает).

Когда интерфейс пытается обслуживать несколько типов клиентов одновременно, как это делает `ServiceContainerInterface`, он нарушает принцип *разделения интерфейса*. Такой интерфейс недостаточно узкоспециализирован, чтобы предназначаться для конкретного клиента.

Рефакторинг: отдельные интерфейсы и множественное наследование

Один из типов клиента – это часть приложения, которая загружает контейнер служб путем настройки доступных сервисов. Такому клиенту потребуется только та часть `ServiceContainerInterface`, ко-

торая делает его изменяемым, то есть метод `set()`. Другим типом клиента является, например, контроллер, который выбирает службу для обработки запроса. Этому типу клиента нужен только метод `get()`, а не `set()`. Разница между клиентами должна отражаться в доступных интерфейсах, например путем разделения интерфейса на `MutableServiceContainerInterface` и `ServiceLocatorInterface`, как показано в листинге 4-3.

Листинг 4-3. Разделение интерфейса

```
interface MutableServiceContainerInterface
{
public function set(string $name, callable $factory): void;
}
interface ServiceLocatorInterface
{
public function get(string $name): object;
}
```

Теперь каждому клиенту может потребоваться собственный соответствующий тип контейнера служб. На практике существует один класс `ServiceContainer`, который обслуживает оба типа клиентов одновременно, реализуя и `MutableServiceContainerInterface`, и `ServiceLocatorInterface` (*см.* листинг 4-4 и рис. 4-1).

Листинг 4-4. Класс ServiceContainer по-прежнему может реализовывать оба интерфейса

```
class ServiceContainer implements
MutableServiceContainerInterface,
ServiceLocatorInterface
{
public function set(string $name, callable $factory): void
{
// ...
}
public function get(string $name): object
{
// ...
}
 }
```

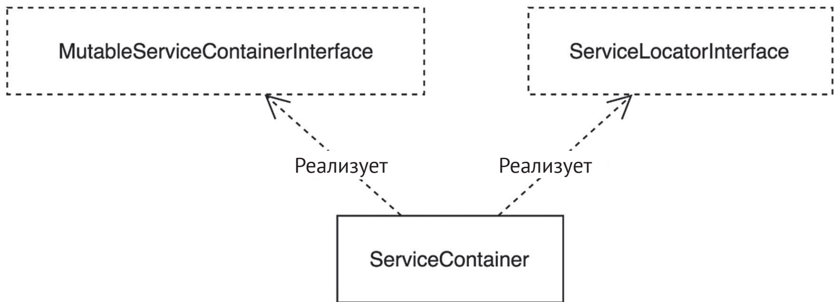


Рис. 4-1. Иерархия класса ServiceContainer

Это не должно беспокоить ни одного из клиентов, поскольку ни один из них не будет зависеть от класса `ServiceContainer`, а только от одного из интерфейсов, как показано в листинге 4-5.

Листинг 4-5. Большинство клиентов будут зависеть только от ServiceLocatorInterface

```
class Kernel
{
public function initializeServiceContainer(
MutableServiceContainerInterface $serviceContainer
) {
$serviceContainer->set(...);
}
}
class SomeController
{
private $serviceLocator;
public function __construct(
ServiceLocatorInterface $serviceLocator
) {
$this->serviceLocator = $serviceLocator;
}
public function indexAction(): Response
{
$mailer = $this->serviceLocator->get('mailer');
// ...
}
}
```

Наличие класса `ServiceContainer` для реализации обоих интерфейсов не является строго необходимым. Просто это облегчит поддержку кода. Смысл в том, что не имеет значения, строго ли класс

следует принципу *разделения интерфейса.* Это не проблема, если все части приложения зависят только от одной небольшой, предназначенной для конкретного клиента части общедоступного API этого класса. Чтобы клиенты могли это делать, убедитесь, что вы всегда предлагаете интерфейс. Затем разделяйте этот интерфейс всякий раз, когда замечаете, что разные клиенты, как правило, используют разные подмножества его методов.

Нарушение: никакого интерфейса, просто класс

Предположим, что вы работаете над пакетом `FabulousORM`, который должен содержать более подходящее средство объектно-реляционного отображения, по сравнению с любым из существующих. Вы определяете класс `EntityManager`, который можно использовать для сохранения сущностей (объектов) в реляционной базе данных (*см.* листинг 4-6). Он использует единицу работы[1] для расчета фактических изменений, которые должны быть внесены в базу данных. У класса `EntityManager` есть несколько открытых методов – `persist()` и `flush()` – и один закрытый метод, который внутренне делает объект `UnitOfWork` доступным для других методов.

Листинг 4-6. Класс EntityManager

```
class EntityManager
{
public function persist(object $entity): void
{
// ...
}
public function flush(): void
{
// ...
}
private function getUnitOfWork(): UnitOfWork
{
// ...
}
}
```

Те, кто используют ваш пакет в своих проектах, могут зависеть от `EntityManager` в своих собственных классах, как это делает `UserRepository` (*см.* листинг 4-7).

[1] *Martin Fowler.* Unit of Work // https://martinfowler.com/eaaCatalog/unitOfWork.html.

Листинг 4-7. Класс UserRepository

```
class UserRepository
{
public function __construct(EntityManager $entityManager)
{
// ...
}
}
```

К сожалению, мы не можем использовать интерфейс в качестве типа для аргумента конструктора `$entityManager`. Класс `EntityManager` не реализует интерфейс. Поэтому лучшее, что мы можем сделать, – это использовать сам класс в качестве типа аргумента конструктора.

Несмотря на то что для класса `EntityManager` нет *явного* интерфейса, у него все равно есть *неявный* интерфейс. Каждый метод класса имеет определенную область видимости (`public`, `protected` или `private`). Когда такой клиент, как `UserRepository`, зависит от класса `EntityManager`, он зависит от всех открытых методов `EntityManager`: `persist()` и `flush()`. Ни один из методов с другой областью видимости (т. е. `protected` или `private`) не может быть вызван клиентом. Таким образом, комбинированные открытые методы образуют *неявный интерфейс* `EntityManager`.

Неявные изменения в неявном интерфейсе

Однажды вы решили добавить класс `Query` в свой пакет ORM. Его можно использовать для выполнения запроса к базе данных и извлечения из нее сущностей. Классу `Query` нужен объект `UnitOfWork`, который используется внутри `EntityManager`. Поэтому вы решаете превратить его закрытый метод `getUnitOfWork()` в открытый. Таким образом, класс `Query` может зависеть от класса `EntityManager` и использовать свой метод `getUnitOfWork()`, как показано в листинге 4-8.

Листинг 4-8. Query нужен только класс EntityManager для объекта UnitOfWork

```
class EntityManager
{
// ...
/**
* Этот метод должен быть открытым, потому что он используется классом
Query;
*/
```

```
public function getUnitOfWork(): UnitOfWork
{
// ...
}
}
class Query
{
public function __construct(EntityManager $entityManager)
{
$this->entityManager = $entityManager;
}
public function someMethod()
{
$this->entityManager->getUnitOfWork()->...
}
}
```

Новый открытый метод – `getUnitOfWork()` – автоматически станет частью *неявного интерфейса* `EntityManager`. С этого момента все клиенты `EntityManager` неявно *зависят* и от этого метода, даже если они могут использовать только методы `persist()` и `flush()`.

Такая ситуация опасна. Возможно, некоторые клиенты тоже начинают использовать общедоступный метод `getUnitOfWork()`. Они могут делать довольно сумасшедшие вещи, используя единицу работы, которые вы обычно вряд ли бы позволили.

Добавление методов в неявный интерфейс класса также может вызвать проблемы обратной совместимости. Скажем, однажды вы проведете рефакторинг класса `Query` и удалите его зависимость от класса `EntityManager`. Так как ни один из ваших классов больше не нуждается в открытом методе `getUnitOfWork()`, вы решаете снова сделать этот метод закрытым. Внезапно все клиенты, которые используют ранее открытый метод `getUnitOfWork()`, выйдут из строя.

Рефакторинг: добавление интерфейсов заголовка и роли

Можно решить эту проблему, определив интерфейс для каждого варианта использования, который предоставляет класс `EntityManager`. Например, можно определить основной вариант использования «постоянных сущностей» например `PersistsEntitiesInterface`, и ввести второй интерфейс, `HasUnitOfWorkInterface`, чтобы определить второй вариант использования (*см.* листинг 4-9).

Листинг 4-9. Используйте заголовок и ролевые интерфейсы

```
interface PersistsEntitiesInterface
{
public function persist(object $entity): void;
public function flush(): void;
}
interface HasUnitOfWorkInterface
{
public function getUnitOfWork(): UnitOfWork;
}
```

Затем вы можете добавить основной интерфейс, который объединяет оба интерфейса, и класс, который реализует основной интерфейс (*см.* листинг 4-10 и рис. 4-2).

Листинг 4-10. Интерфейс заголовка для EntityManager объединяет все его интерфейсы ролей

```
interface EntityManagerInterface extends
PersistsEntitiesInterface,
HasUnitOfWorkInterface
{
}
class EntityManager implements EntityManagerInterface
{
// ...
}
```

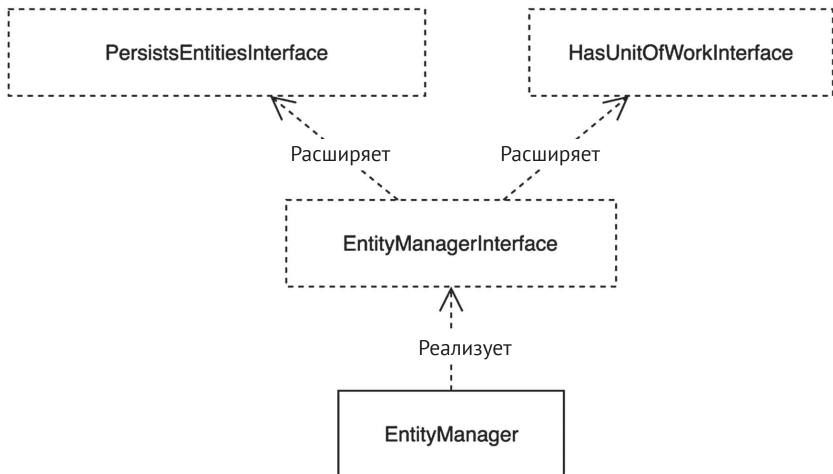


Рис. 4-2. Иерархия класса EntityManager

Несколько интерфейсов, которые мы только что определили, описывают *роли,* которые может играть класс: `PersistsEntitiesInterface` и `HasUnitOfWorkInterface`. Затем идет один интерфейс, который объединяет эти роли, и все вместе это образует то, что известно как *диспетчер сущностей*, который «может сохранять сущности» и «имеет единицу работы»: `EntityManagerInterface`.

Мартин Фаулер называет эти разные типы интерфейсов интерфейсами ролей и интерфейсами заголовков[1] соответственно. Вы можете определить интерфейсы ролей для класса, посмотрев на разных клиентов, которые используют этот класс. Затем вы группируете методы, которые используются вместе в отдельные интерфейсы, как мы делали для класса `EntityManager`.

Интерфейсы заголовка обычно проще всего определить, поскольку:

> *все, что вам нужно сделать, – это продублировать открытые методы [класса], без раздумий*[2].

Часто определения одного только интерфейса заголовка недостаточно, как в случае с `EntityManager`. Клиентам не понадобятся никакие другие открытые методы, кроме `persist()` и `flush()`. Если интерфейс заголовка может содержать другие открытые методы, такие как `getUnitOfWork()`, они будут лишними. Как говорит Роберт С. Мартин[3]:

> *Клиентов не следует заставлять зависеть от методов, которые они не используют.*

Пакеты и принцип разделения интерфейса

Для разработчиков пакетов применение принципа *разделения интерфейса* имеет несколько преимуществ. Во-первых, это приведет к меньшим по размеру интерфейсам, которые актуальны для подмножества всех клиентов. Как мы видели ранее при обсуждении принципа *единственной ответственности*, уменьшение класса (или интерфейса) уменьшит количество причин для его измене-

---

1 *Martin Fowler*. RoleInterface // https://martinfowler.com/bliki/RoleInterface.html.

2 *Там же*.

3 *Robert C. Martin*. The Interface Segregation Principle // Engineering Notebook. C++ Report, Nov-Dec, 1996 (PDF доступен по адресу http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod).

ния. Интерфейс, который нужно менять реже, гораздо предпочтительнее, поскольку вам будет проще поддерживать обратную совместимость. В качестве примера можно привести `ServiceLocatorInterface`, который получился у нас ранее в этой главе, – наличие только метода `get ($id)` делает интерфейс очень стабильным и с меньшей вероятностью изменения.

Во-вторых, когда вы добавляете небольшой, сфокусированный интерфейс к классу в своем пакете, вы можете добавить к этому классу дополнительные открытые методы, которые не являются частью опубликованного интерфейса.

Вы даже можете изменить или удалить существующие методы, которые не являются частью интерфейса, предоставляя себе больше свободы для перепроектирования или рефакторинга классов, не мешая пользователям.

Я повторюсь, что не каждому классу в пакете действительно нужен интерфейс. Мы уделим немного времени обсуждению ряда правил, когда следует и когда не следует добавлять интерфейс к классу, в конце главы 5.

Заключение

Интерфейс обычно имеет несколько методов, хотя не каждый клиент интерфейса использует одно и то же подмножество этих методов. В зависимости от интерфейсов клиент также будет неявно зависеть от всех неиспользуемых методов. Принцип *разделения интерфейса* велит нам разделять (сегрегировать) методы интерфейса в соответствии с тем, как они используются.

Разделение интерфейса может произойти путем разграничения конкретных ролей более крупного интерфейса. В итоге вы получите интерфейс «заголовка» и несколько «ролевых» интерфейсов. Иногда общей концепции не существует. В этом случае разделение интерфейса приведет к появлению нескольких независимых интерфейсов.

Если у класса нет явного («опубликованного») интерфейса, набор открытых методов, которые он предлагает, считается его интерфейсом. Чтобы ограничить число открытых методов, от которых должен зависеть клиент для использования этого класса, для начала нужно опубликовать интерфейс для этого класса. Затем можно применить принцип *разделения интерфейса* и сделать интерфейс меньше или разделить его на отдельные интерфейсы для каждого набора клиентов.

Глава 5

Принцип инверсии зависимостей

Последний из принципов разработки классов SOLID фокусируется на *зависимостях* классов. Он говорит вам, от чего должен зависеть класс[1]:

> *Зависеть от абстракций, а не от конкретных реализаций.*

Название этого принципа содержит слово «инверсия», из чего можно сделать вывод, что без следования этому принципу мы бы зависели от конкретных реализаций, а не от абстракций. Данный принцип велит нам изменить это направление: мы всегда должны зависеть от абстракций.

Пример инверсии зависимостей: генератор FizzBuzz

Существует хорошо известное задание по программированию, которое служит хорошим примером инверсии зависимостей. Оно называется «FizzBuzz» и часто используется в качестве небольшого теста, чтобы увидеть, сможет ли кандидат, претендующий на должность программиста, выполнить ряд требований, обычно на месте. Вот как выглядят эти требования:

- создайте список чисел от 1 до n;
- числа, которые делятся на 3, следует заменить на Fizz;
- числа, которые делятся на 5, должны быть заменены на Buzz;
- числа, которые делятся на 3 *и* на 5, должны быть заменены на FizzBuzz.

---

[1] *Robert C. Martin.* The principles of OOD // http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

Если применить эти правила, мы получим такой список:

1, 2, Fizz, 4, Buzz … 13, 14, FizzBuzz, 16, 17 …

Поскольку не все элементы списка являются целыми числами, получившийся список должен быть списком строк. Простая реализация может выглядеть так, как показано в листинге 5-1.

Листинг 5-1. Реализация алгоритма FizzBuzz

```
class FizzBuzz
{
public function generateList(int $limit): array
{
$list = [];
for ($number = 1; $number <= $limit; $number++) {
$list[] = $this->generateElement($number);
}
return $list;
}
private function generateElement(int $number): string
{
if ($number % 3 === 0 && $number % 5 === 0) {
return 'FizzBuzz';
}
if ($number % 3 === 0) {
return 'Fizz';
}
if ($number % 5 === 0) {
return 'Buzz';
}
return (string)$number;
}
}
$fizzBuzz = new FizzBuz();
$list = $fizzBuzz->generateList(100);
```

Учитывая задание, это очень точная реализация требований. Читая код, мы можем распознать в нем каждое требование: правила о делимости чисел, требование, чтобы список чисел начинался с 1, и т. д. После того как кандидат подготовил такой код, интервьюер добавляет еще одно требование:

> *Должна быть возможность добавить дополнительное правило без изменения класса* FizzBuzz.

Делаем класс FizzBuzz открытым для расширения

В настоящее время класс `FizzBuzz` не открыт для расширения и не закрыт для модификации. Если числа, делимые на 7, в один прекрасный день будут заменены на Whiz, это изменение будет невозможно реализовать без фактического изменения кода класса `FizzBuzz`.

Размышляя о дизайне класса `FizzBuzz` и о том, как можно сделать его более гибким, отметим, что метод `generateElement()` содержит много деталей. Однако в рамках того же класса метод `generateList()` довольно универсален. Он просто генерирует список увеличивающихся чисел, начиная с 1 (что несколько специфично) и заканчивая данным числом. Таким образом, у класса `FizzBuzz` есть две ответственности: он генерирует списки чисел и заменяет определенные числа чем-то другим, основываясь на правилах FizzBuzz.

Эти правила FizzBuzz подлежат изменению. И требование заключается в том, что при изменении правил нам не нужно изменять сам класс `FizzBuzz`. Итак, давайте применим то, что мы узнали в главе, посвященной принципу *открытости/закрытости*. Для начала мы можем извлечь правила в их собственные классы и использовать их в `generateElement()`, как показано в листинге 5-2.

Листинг 5-2. Извлечение метода для генерации отдельных элементов с использованием «правил»

```
class FizzBuzz
{
public function generateList(int $limit): array
{
// ...
}
private function generateElement(int $number): string
{
$fizzBuzzRule = new FizzBuzzRule();
if ($fizzBuzzRule->matches($number)) {
return $fizzBuzzRule->getReplacement();
}
$fizzRule = new FizzRule();
if ($fizzRule->matches($number)) {
return $fizzRule->getReplacement();
}
```

```
$buzzRule = new BuzzRule();
if ($buzzRule->matches($number)) {
return $buzzRule->getReplacement();
}
return (string)$number;
}
}
```

Подробности о правилах можно найти в конкретных классах правил. В листинге 5-3 показан пример правила «Fizz», реализованного в классе `FizzRule`.

Листинг 5-3. Класс, представляющий одно из правил FizzBuzz

```
class FizzRule
{
public function matches($number): bool
{
return $number % 3 === 0;
}
public function getReplacement(): string
{
return 'Fizz';
}
}
```

Это один из шагов в правильном направлении. Даже несмотря на то, что сведения о правилах (числа 3, 5, 3 и 5 и их замещающие значения) были перенесены в конкретные классы правил, код в `generateElement()` остается очень специфичным. Правила по-прежнему представлены (очень специфическими) именами классов, и добавление нового правила все равно потребует модификации метода `generateElement()`, поэтому мы еще не сделали класс *открытым для расширения*.

Избавляемся от специфичности

Мы можем убрать эту специфичность из класса `FizzBuzz`, используя интерфейс (*см.* листинг 5-4) для классов правил и допуская внедрение нескольких правил в экземпляр `FizzBuzz`.

Листинг 5-4. Внедрение абстракции

```
interface RuleInterface
{
```

```
public function matches($number): bool;
public function getReplacement(): string;
}
class FizzBuzz
{
private $rules = [];
public function addRule(RuleInterface $rule): void
{
$this->rules[] = $rule;
}
public function generateList($limit): array
{
// ...
}
private function generateElement(int $number): string
{
foreach ($this->rules as $rule) {
if ($rule->matches($number)) {
return $rule->getReplacement();
}
}
return $number;
}
}
```

Теперь нам нужно убедиться, что каждый конкретный класс правил реализует `RuleInterface`, а затем класс `FizzBuzz` можно использовать для генерации списков чисел с различными правилами, как показано в листинге 5-5 и на рис. 5-1.

Листинг 5-5. Настройка экземпляра FizzBuzz с конкретными правилами

```
class FizzRule implements RuleInterface
{
// ...
}
$fizzBuzz = new FizzBuzz();
$fizzBuzz->addRule(new FizzBuzzRule());
$fizzBuzz->addRule(new FizzRule());
$fizzBuzz->addRule(new BuzzRule());
// add more rules if you want, e.g.
// $fizzBuzz->addRule(new WhizzRule());
// ...
$list = $fizzBuzz->generateList(100);
```

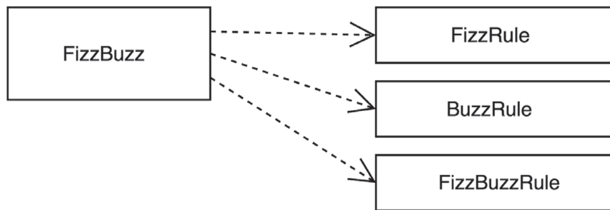


Рис. 5-1. FizzBuzz с конкретными зависимостями

Теперь у нас есть весьма универсальный фрагмент кода, класс `FizzBuzz`, который «генерирует список чисел, заменяя определенные числа строками на основе гибкого набора правил». В этом описании нет упоминания о «FizzBuzz», а в коде класса `FizzBuzz` нет упоминания ни о «Fizz», ни о «Buzz». На самом деле класс `FizzBuzz` можно переименовать, чтобы он лучше информировал о своей ответственности. Конечно, именование является одной из самых сложных частей нашей работы, и `NumberListGenerator` не является особенно выразительным именем, но оно лучше описывает его назначение, чем его текущее имя.

Глядя на первоначальную реализацию класса `FizzBuzz`, становится ясно, что у класса с самого начала была абстрактная задача: создать список чисел. Только правила были очень подробными (делиться на 3, на 5 и т. д.). Говоря словами из принципа *инверсии зависимостей*: абстракция зависит от конкретных вещей. Это привело к закрытию класса `FizzBuzz` для расширения, поскольку добавить другое правило без его изменения было невозможно.

Введя `RuleInterface` и добавив конкретные классы правил, которые реализовали этот интерфейс, мы исправили направление зависимости. Класс `FizzBuzz` начал зависеть от более абстрактных вещей, называемых «правилами» (см. рис. 5-2). При создании нового экземпляра `FizzBuzz` конкретные реализации `RuleInterface` должны внедряться в правильном порядке. Это приведет к правильному выполнению алгоритма FizzBuzz. Сам класс `FizzBuzz` больше не заботится об этом, поэтому класс становится более гибким относительно меняющихся требований. Именно так и должно быть в соответствии с принципом *инверсии зависимостей*[1]:

---

[1] *Robert C. Martin*. (May 1996) The Dependency Inversion Principle // C++ Report (доступно в формате PDF по адресу: http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod).

Абстракции не должны зависеть от деталей. Детали должны зависеть от абстракций.

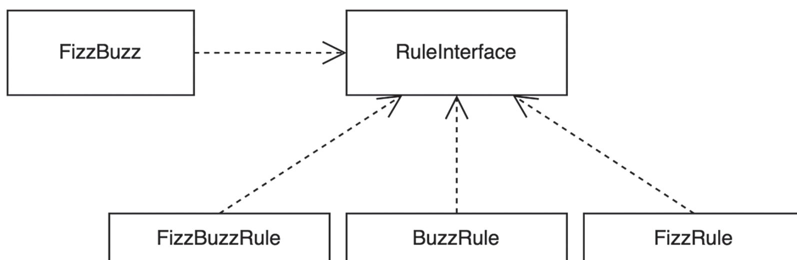


Рис. 5-2. FizzBuzz с абстрактными зависимостями

Теперь, когда мы увидели принцип *инверсии зависимости* в действии, можно рассмотреть ситуации, когда он явно нарушается.

Нарушение: высокоуровневый класс зависит от низкоуровнего

Первое нарушение возникает из-за *смешивания различных уровней абстракции*. Это интересная концепция, которая требует дальнейшего объяснения, прежде чем мы рассмотрим пример подробно.

Ежедневно нам приходится иметь дело с огромным разнообразием вещей, которые существуют во вселенной. Мы не смогли бы сделать что-либо значимое, если бы рассматривали каждую мельчайшую деталь всего, о чем мы говорим, всего, что мы используем при выполнении своей работы, всех, кого любим. Поэтому, чтобы сохранить здравомыслие, мы постоянно придумываем абстракции. «Абстракция» означает «убрать детали». То, что остается, – это концепция, которую можно использовать для *группировки* всех конкретных вещей, из которых была создана абстракция, и название того, что *важно* для всех этих конкретных вещей, игнорируя небольшие различия.

В разговоре мы обычно устанавливаем некий уровень абстракции, поэтому можем спокойно игнорировать детали (или более широкую картину). При обсуждении разработки программного обеспечения это означает, что мы приглядываемся, пока не сможем решить текущую проблему. Например, при обсуждении кода «с запашком» мы будем говорить о сигнатурах методов, именах переменных

и т. д., поэтому можем спокойно игнорировать очереди сообщений или конкретную файловую систему Linux, которую используем на нашем сервере. Когда мы говорим о том, как приложение получает данные из базы данных, мы обсуждаем SQL-запросы, поэтому можем игнорировать основной протокол TCP, который используется.

Когда приглядываемся, это аналогично переходу от абстракции к конкретному и обратно. Чем больше мы приглядываемся к части нашего программного обеспечения, тем ближе мы к низкоуровневым деталям (также известным как «внутренние компоненты»). Чем больше отдаляемся, тем ближе мы подходим к высокоуровневому представлению о системе, какие функции она стремится предоставить своим пользователям.

При проектировании классов мы должны учитывать один и тот же тип увеличения и уменьшения. Каждый класс имеет два уровня абстракции: первый воспринимается клиентами, второй – тот, что происходит внутри. По определению класс или интерфейс будет скрывать некоторые детали реализации для своего клиента, а это означает, что клиенты будут воспринимать его как более абстрактный, в то время как внутри класс более конкретен.

Таким образом, внутренняя часть класса всегда более конкретна, чем абстракция, которую представляет класс. Однако когда класс зависит от какого-либо другого класса, он снова должен зависеть от чего-то абстрактного, а не конкретного. Таким образом, сам класс становится клиентом чего-то абстрактного и может безопасно игнорировать все основные детали того, как работает эта зависимость.

В качестве примера класса, который имеет зависимость, не являющуюся абстрактной, давайте рассмотрим класс `Authentication` в листинге 5-6.

Листинг 5-6. Класс Authentication

```
class Authentication
{
private $connection;
public function __construct(Connection $connection)
{
$this->connection = $connection;
}
public function checkCredentials(
string $username,
string $password
): void {
```

```
$user = $this->connection->fetchAssoc(
'SELECT * FROM users WHERE username = ?',
[$username]
);
if ($user === null) {
throw new InvalidCredentialsException(
'User not found'
);
}
// validate password
}
}
```

Класс `Authentication` нуждается в соединении с базой данных (*см*. рис. 5-3), которое в этом случае представлено объектом `Connection`. Он использует соединение для извлечения пользовательских данных из базы данных.

Рис. 5-3. Класс Authentication зависит от Connection

У этого подхода много проблем. Их можно сформулировать, ответив на приведенные ниже вопросы об этом классе.

1. *Механизму аутентификации важно иметь дело с точным местоположением пользовательских данных?*

   Ну, определенно нет. Единственное, что действительно нужно классу `Authentication`, – это пользовательские данные в виде массива или, предпочтительно, объекта, обозначающего пользователя. *Происхождение* этих данных не имеет значения.

2. *Можно ли получить пользовательские данные из другого места, кроме базы данных?*

   В настоящее время это невозможно. Классу `Authentication` требуется объект `Connection`, который является соединением с *базой данных*. Его нельзя использовать для извлечения пользователей, например, из текстового файла или какого-либо внешнего веб-сервиса.

Глядя на эти ответы, мы должны сделать вывод, что в этом классе были нарушены и принцип *единственной ответственности*, и принцип *открытости/закрытости*. Класс `Authentication` касается не только самого механизма аутентификации, но и фактического

хранения пользовательских данных. Кроме того, невозможно перенастроить класс, чтобы искать пользовательские данные в другом месте. Основная причина этих проблем заключается в том, что принцип *инверсии зависимостей* также был нарушен: сам класс `Authentication` является высокоуровневой абстракцией. Тем не менее он зависит от конкретной реализации очень низкого уровня: соединения с базой данных. Эта конкретная зависимость делает невозможным для класса `Authentication` извлечение пользовательских данных из любого другого места, кроме базы данных.

Пытаясь перефразировать то, что действительно нужно классу `Authentication`, мы понимаем, что это не соединение с базой данных, а просто нечто, что может *предоставить* пользовательские данные. Давайте назовем это «поставщиком пользователей». Класс `Authentication` не должен ничего знать о реальном процессе извлечения пользовательских данных (независимо от того, происходят ли они из базы данных, текстового файла или сервера LDAP). Ему лишь нужны эти пользовательские данные.

Было бы хорошо, если бы класс `Authentication` не заботился о происхождении самих пользовательских данных. Все детали реализации по извлечению пользовательских данных могут быть исключены из этого класса. Класс сразу же станет многократно используемым, потому что пользователи класса смогут реализовать своих собственных «поставщиков пользователей».

Рефакторинг: абстракции и конкретные реализации зависят от абстракций

Рефакторинг высокоуровневого класса `Authentication`, чтобы он следовал принципу *инверсии зависимостей*, означает, что для начала мы должны удалить зависимость от низкоуровневого класса `Connection`. Затем мы добавляем высокоуровневую зависимость от того, что предоставляет пользовательские данные класса `UserProvider` (*см.* листинг 5-7).

Листинг 5-7. Класс UserProvider

```
class Authentication
{
private $userProvider;
public function __construct(UserProvider $userProvider)
{
```

```
$this->userProvider = $userProvider;
}
public function checkCredentials(
string $username,
string $password
): void {
$user = $this->userProvider->findUser($username);
if ($user === null) {
throw new InvalidCredentialsException(
'User not found’
);
}
// validate password
}
}
class UserProvider
{
private $connection;
public function __construct(Connection $connection)
{
$this->connection = $connection;
}
public function findUser(string $username): array
{
return $this->connection->fetchAssoc(
'SELECT * FROM users WHERE username = ?',
[$username]
);
}
}
```

Класс `Authentication` больше не имеет ничего общего с базой данных (как показано на рис. 5-4). Класс `UserProvider` делает все, что нужно для извлечения пользователя из базы данных.

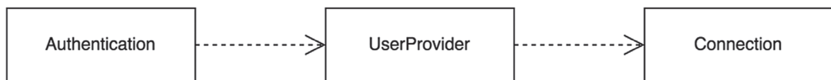


Рис. 5-4. Authentication зависит от UserProvider

Переключаться между реализациями различных поставщиков пользователей по-прежнему непросто. Класс `Authentication` зависит от конкретного класса `UserProvider`. Если кто-то хочет получить свои пользовательские данные из текстового файла, ему придется расширить этот класс и переопределить его метод `findUser()` (как это было сделано в листинге 5-8).

Листинг 5-9. UserProviderInterface и некоторые реализации

```
interface UserProviderInterface
{
public function findUser(string $username): array;
}
class MySQLUserProvider implements UserProviderInterface
{
// ...
}
class TextFileUserProvider implements UserProviderInterface
{
// ...
}
```

И конечно же, мы должны изменить тип аргумента конструктора класса `Authentication` на `UserProviderInterface` (*см.* листинг 5-10).

Листинг 5-10. Класс Authentication теперь принимает UserProviderInterface

```
class Authentication
{
private $userProvider;
public function __construct(
UserProviderInterface $userProvider
) {
$this->userProvider = $userProvider;
}
// ...
}
```

Как видно на диаграмме зависимостей на рис. 5-5, высокоуровневый класс `Authentication` больше не зависит от низкоуровневых конкретных классов, таких как `Connection`. Теперь он зависит от интерфейса: `UserProviderInterface`. Оба они концептуально находятся на более или менее одинаковом уровне. Операции более низкого уровня, такие как чтение из файла и выборка данных из базы данных, выполняются классами более низшего уровня – конкретными поставщиками пользователей. Это полностью соответствует принципу *инверсии зависимостей*, который гласит:

> *Модули верхних уровней не должны зависеть от модулей нижних уровней. Оба типа модулей должны зависеть от абстракций*[1].

---

[1] *Robert C. Martin*. (May 1996) The Dependency Inversion Principle. C++ Report (доступно в формате PDF по адресу: http://www.butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod).

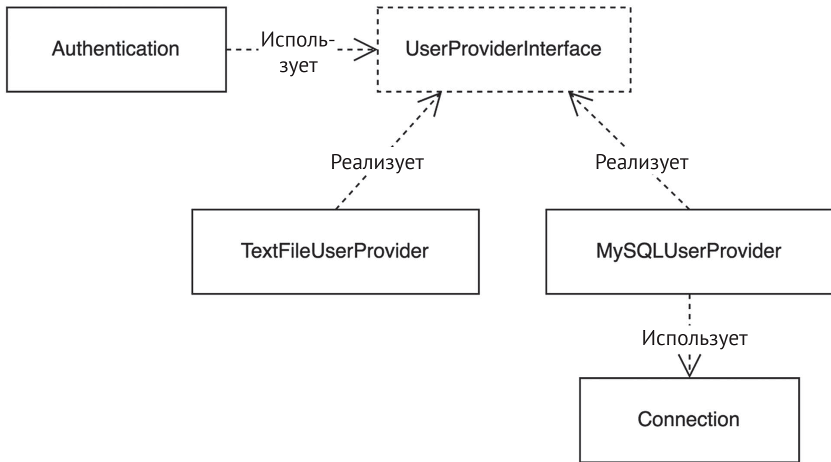


Рис. 5-5. Authentication зависит от UserProviderInterface

Приятным побочным эффектом внесенных нами изменений является то, что удобство сопровождения кода значительно улучшилось. Когда в одном из запросов, используемых для извлечения пользовательских данных из базы данных, обнаруживается ошибка, больше не нужно изменять сам класс `Authentication`. Необходимые изменения будут происходить только внутри конкретного поставщика пользователей, в этом случае `MySQLUserProvider`. Это означает, что этот рефакторинг значительно уменьшил вероятность того, что вы случайно сломаете сам механизм аутентификации.

ПРОСТО ЗАВИСИТЬ ОТ ИНТЕРФЕЙСА НЕДОСТАТОЧНО

Переход от `UserProvider` к `UserProviderInterface` был важным, потому что он помог пользователям класса `Authentication` легко переключаться между реализациями поставщиков пользователей. Но просто добавить интерфейс к классу не всегда достаточно, чтобы исправить все проблемы, связанные с зависимостями.

Рассмотрим альтернативную версию `UserProviderInterface`, показанную в листинге 5-11.

Листинг 5-11. Альтернативный интерфейс UserProviderInterface

```
interface UserProviderInterface
{
```

```
public function findUser(string $username): array;
public function getTableName(): string;
}
```

Это не очень полезный интерфейс. Он нарушает принцип *подстановки Лисков*. Не все классы, которые реализуют этот интерфейс, смогут стать хорошей заменой ему. Если одна реализация не использует таблицу базы данных для хранения пользовательских данных, она наверняка не сможет вернуть правильное значение, когда кто-то вызовет для нее метод `getTableName()`. Но что еще более важно: `UserProviderInterface` смешивает различные уровни абстракции, сочетая нечто высокоуровневое, например «поиск пользователя», и нечто низкоуровневое, например «имя таблицы базы данных».

Таким образом, даже если бы мы ввели этот интерфейс, чтобы класс `Authentication` зависел от абстракции, а не от конкретной реализации, эта цель не будет достигнута. Фактически класс `Authentication` по-прежнему будет зависеть от чего-то конкретного и низкоуровневого: поставщика пользователей на базе таблицы.

Нарушение: привязка к поставщику

В этом разделе мы обсудим распространенное нарушение принципа *инверсии зависимостей*, которое особенно важно для разработчиков пакетов. Скажем, классу нужен какой-то способ для запуска событий на уровне приложения в целом. Обычным решением для этого является использование диспетчера событий. Проблема состоит в том, что диспетчеров событий, которые доступны, много, и у них у всех слегка разный API. Например, `EventDispatcherInterface`[1] из фреймворка Symfony выглядит так, как показано в листинге 5-12.

Листинг 5-12. EventDispatcherInterface

```
interface EventDispatcherInterface
{
public function dispatch(
string $eventName,
Event $event = null
): void;
public function addListener(
string $eventName,
callable $listener,
```

[1] https://github.com/symfony/event-dispatcher/blob/2.3/EventDispatcherInterface.php.

```
int $priority = 0
);
// ...
}
```

Обратите внимание, что события должны иметь имя, которое является строкой (например, "new_user "), и при запуске (или «отправке») события вы можете предоставить объект Event, содержащий дополнительные контекстные данные. Объект будет дополнен и использован в качестве первого аргумента, когда слушатель событий (который может быть любым типом callable) получит уведомление.

Пример класса Event и слушателя событий можно найти в листинге 5-13.

Листинг 5-13. Класс Event и слушатель событий

```
use Symfony\Component\EventDispatcher\Event;
class NewUserEvent extends Event
{
private $user;
public function __construct(User $user)
{
$this->user = $user;
}
public function getUser(): User
{
return $this->user;
}
}
class EventListener
{
public function onNewUser(NewUserEvent $event): void
{
// ...
}
}
$eventDispatcher = new EventDispatcher();
$eventDispatcher->addListener(
'new_user',
[new EventListener(), 'onNewUser']
);
$user = new User();
$eventDispatcher->dispatch('new_user', new NewUserEvent($user))
```

Диспетчер событий из другого фреймворка, Laravel, похож на тот, что показан в листинге 5-14 (на базе версии 4.0[1]).

1 https://github.com/laravel/framework/blob/4.0/src/Illuminate/Events/Dispatcher.php.

Листинг 5-14. Диспетчер событий из Laravel

```
class Dispatcher
{
public function listen(
string $event,
callable $listener,
int $priority = 0
): void {
...
}
public function fire(string $event, array $payload = []): void
{
// ...
}
// ...
}
```

Обратите внимание, что он не реализует интерфейс. И вместо объекта события контекстные данные для событий («полезная нагрузка») состоят из массива, который будет использоваться в качестве аргумента метода, когда слушатель получит уведомление о событии. *См.* листинг 5-15, где приводится пример того, как это используется.

Листинг 5-15. Использование диспетчера событий Laravel

```
class EventListener
{
public function onNewUser(User $user)
{
// ...
}
}
$dispatcher = new Dispatcher();
$dispatcher->listen(
'new_user',
[new EventListener(), 'onNewUser']
);
$user = new User();
$dispatcher->fire('new_user', [$user]);
```

Похоже, что вы можете делать более или менее одинаковые действия с обоими диспетчерами событий, то есть запускать события и прослушивать их. Но то, как вы это делаете, отличается.

Допустим, пакет, над которым вы работаете, содержит класс `UserManager`, как тот, что мы видели в листинге 5-16. Используя этот

класс, вы можете создавать новых пользователей. После этого вам нужно отправить событие на уровне приложения в целом, чтобы другие части приложения могли отреагировать на тот факт, что теперь существует новый пользователь (например, возможно, новые пользователи должны получить приветственное письмо).

Листинг 5-16. Класс UserManager

```
use Illuminate\Events\Dispatcher;
class UserManager
{
public function create(User $user): void
{
// Обеспечиваем хранение пользовательских данных;
// ...
// Инициируем событие: "new_user"
}
}
```

Предположим, вы хотите использовать пакет, содержащий класс `UserManager`, в приложении, написанном на Laravel. Laravel уже предоставляет экземпляр класса `Dispatcher` в своем IoC-контейнере. Это означает, что вы можете легко вставить его в качестве аргумента конструктора класса `UserManager`, как показано в листинге 5-17.

Листинг 5-17. Использование диспетчера событий Laravel в UserManager

```
use Illuminate\Events\Dispatcher;
class UserManager
{
private $dispatcher;
public function __construct(Dispatcher $dispatcher)
{
$this->dispatcher = $dispatcher;
}
public function create(User $user): void
{
// ...
$this->dispatcher->fire('new_user', ['user' => $user]);
}
}
```

Пару недель спустя вы начинаете работать над проектом, созданным с использованием фреймворка Symfony. Вы хотите повторно использовать класс `UserManager`, поскольку он предлагает именно ту функциональность, которая вам нужна, и вы устанавливаете пакет, где он содержится, внутри нового проекта. У при-

ложения, написанного на Symfony, также есть диспетчер событий, легкодоступный в его контейнере служб. Но этот диспетчер событий является экземпляром `EventDispatcherInterface`. *Невозможно* использовать диспетчер событий Symfony в качестве аргумента конструктора для класса `UserManager`, поскольку тип аргумента не будет соответствовать типу внедренной службы. Вы эффективно *предотвратили повторное использование* класса `UserManager`.

Если вы все еще хотите использовать класс `UserManager` в проекте Symfony, вам нужно будет добавить дополнительную зависимость от пакета `litate/events`, чтобы сделать класс `Dispatcher` доступным в вашем проекте. Вам нужно настроить службу для нее рядом с уже существующим диспетчером событий Symfony, и в итоге получится два глобальных диспетчера событий. После этого вам по-прежнему нужно будет сократить разрыв между обоими типами диспетчеров, поскольку события, запускаемые в `Dispatcher` Laravel, не будут запускаться автоматически и в диспетчере событий Symfony. Фактически они даже используют несовместимые типы (объекты событий и массивы).

В тот момент, когда вы выбрали диспетчер событий Laravel, вы связали пакет с конкретной реализацией, затрудняя или делая невозможным использование пакета в проекте, который использует другой диспетчер событий. Внедрение такой зависимости в ваш пакет называется «привязкой к поставщику»; он будет работать только со сторонним кодом от конкретного поставщика.

Решение: добавляем абстракцию и удаляем зависимость с помощью композиции

Как мы уже обсуждали ранее, зависимость от конкретного класса сама по себе может быть проблематичной, поскольку пользователям трудно переключаться между реализациями этой зависимости. Поэтому мы должны представить собственный интерфейс, который отделяет этот класс от любой конкретной реализации диспетчера событий. Этот абстрактный диспетчер событий не зависит от фреймворка, он просто предлагает один из методов, который можно использовать для отправки событий. Затем мы можем изменить класс `UserManager` таким образом, чтобы он принимал только диспетчер событий, который является экземпляром нашего собственного интерфейса `DispatcherInterface` (*см.* листинг 5-18).

Листинг 5-18. Абстракция и ее использование в UserManager

```
interface DispatcherInterface
{
public function dispatch($eventName, array $context = []);
}
class UserManager
{
private $dispatcher;
public function __construct(DispatcherInterface $dispatcher)
{
$this->dispatcher = $dispatcher;
}
// ...
}
```

Класс `UserManager` теперь полностью отделен от фреймворка. Он использует собственный диспетчер событий, который является достаточно общим и содержит минимальное количество деталей.

Конечно, наш интерфейс `DispatcherInterface` сам по себе не является рабочим диспетчером событий. Нам необходимо преодолеть разрыв между этим интерфейсом и диспетчерами конкретных событий из Laravel и Symfony. Это можно сделать с помощью шаблона *Адаптер*[1]. Используя композицию объектов, мы можем сделать класс `Dispatcher` совместимым с нашим собственным `DispatcherInterface`, как показано в листинге 5-19.

Листинге 5-19. Конкретная реализация абстрактного `DispatcherInterface`, использующего диспетчер событий Laravel

```
use Illuminate\Events\Dispatcher;
class LaravelDispatcher implements DispatcherInterface
{
private $dispatcher;
public function __construct(Dispatcher $dispatcher)
{
$this->dispatcher = $dispatcher;
}
public function dispatch(
string $eventName,
array $context = []
): void {
$this->dispatcher->fire(
```

[1] *Erich Gamma e. a.* Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

```
$eventName,
array_values($context)
);
}
}
```

Внедрив `DispatcherInterface`, мы очистили путь для пользователей других фреймворков, чтобы они могли реализовать собственные классы-адаптеры. Эти классы должны соответствовать только общедоступному API, определенному нашим интерфейсом `DispatcherInterface`. Под капотом они могут использовать свой собственный тип диспетчера событий. Например, адаптер для диспетчера событий Symfony будет выглядеть так, как показано в листинге 5-20.

Листинг 5-20. Альтернативная реализация DispatcherInterface, использующая диспетчер событий Symfony

```
use Symfony\Component\EventDispatcher\EventDispatcherInterface;
use Symfony\Component\EventDispatcher\GenericEvent;
class SymfonyDispatcher implements DispatcherInterface
{
private $dispatcher;
public function __construct(
EventDispatcherInterface $dispatcher
) {
$this->dispatcher = $dispatcher;
}
public function dispatch(
string $eventName,
array $context = []
): void {
$this->dispatcher->dispatch(
$eventName,
new GenericEvent(null, $context)
);
}
}
```

До того, как мы ввели `DispatcherInterface`, UserManager зависел от кое-чего *конкретного* – специфичной для Laravel реализации диспетчера событий, как показано на рис. 5-6.

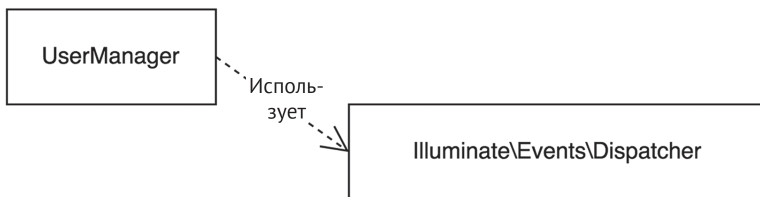


Рис. 5-6. UserManager имеет зависимость от конкретного диспетчера Laravel

После того как мы добавили `DispatcherInterface`, класс `UserManager` теперь зависит от кое-чего *абстрактного*. Другими словами, мы изменили направление зависимости, а это именно то, что велит нам сделать принцип *инверсии зависимостей*. Получившаяся диаграмма зависимости показана на рис. 5-7.

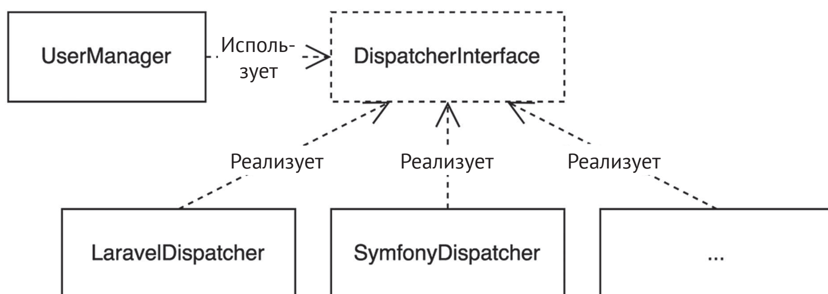


Рис. 5-7. UserManager имеет зависимость от абстрактного диспетчера, от которого зависят несколько адаптеров

Пакеты и принцип инверсии зависимостей

Хотя принцип *инверсии зависимостей* является принципом проектирования классов, он касается *взаимосвязи* между классами. Эта связь часто пересекает границы пакета. Поэтому принцип *инверсии зависимостей* сильно резонирует на уровне пакета. Согласно этому принципу, классы должны зависеть от абстракций, а не от конкретных реализаций. Параллельно с этим сами зависимости должны также быть направлены в сторону абстрактности, как мы увидим в главе 11.

Зависимость от стороннего кода: всегда ли это плохо?

Мы избавились от привязки к поставщику в случае с классом `UserManager` и теперь знаем принцип, с помощью которого можно добиться того же во многих других ситуациях. Однако при этом нужно заплатить определенную цену – цену определения нашего собственного интерфейса и собственных реализаций адаптера. Даже если мы используем инверсию зависимостей для каждой зависимости каждого класса, существуют другие способы, в которых наш код будет оставаться зависимым от стороннего кода.

Таким образом, вопрос заключается в том, в каких случаях мы должны позволять себе зависеть от стороннего кода и какие случаи непременно требуют инверсии зависимостей?

Во-первых, нужно провести различие между фреймворками и библиотеками. Хотя и те, и другие могут распространяться в виде пакетов, разница наиболее очевидна, если учесть, как они работают с вашим кодом. Фреймворки следуют голливудскому принципу[1]: «Не звоните нам, мы сами позвоним вам». Например, веб-сервер может переслать HTTP-запрос вашему веб-приложению, а его фреймворк проанализирует запрос и вызовет один из ваших контроллеров. После того как фреймворк вызвал *ваш* код (также известный как «код, выполняемый в пользовательском пространстве»), вы можете использовать все, что угодно, для выполнения своей задачи, включая код сторонней библиотеки.

На рис. 5-8 показано, как фреймворк, код, выполняемый в пользовательском пространстве, и код библиотеки вызывают друг друга.

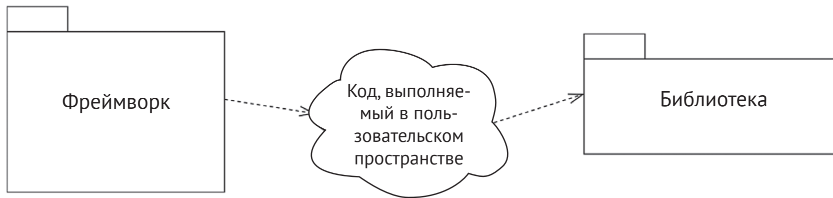


Рис. 5-8. Фреймворк вызывает код в пользовательском пространстве, который вызывает код библиотеки

Если вы разработчик пакета, который хочет извлечь часть кода, выполняемого в пользовательском пространстве, и опубликовать

[1] *Craig Larman.* Applying UML and Patterns. Prentice Hall (2001).

его в качестве пакета, вы должны извлекать только ту часть кода, которая не связана с фреймворком. Убедитесь, что код будет полезен для всех пользователей, независимо от того, какой перед ними фреймворк. Оставшуюся часть кода, которая связана с фреймворком, можно извлечь в пакет, предназначенный для конкретного фреймворка, часто называемый пакетом-«мостом».

Рассматривая независимый от фреймворка код, который теперь извлечен в пакет, вы можете приступить к применению там принципа *инверсии зависимостей* и реализовывать интерфейсы и код адаптера для конкретных классов из библиотек, которые использует ваш пакет. На рис. 5-9 показан граф зависимостей получившихся пакетов, когда был извлечен код пользовательского пространства и были добавлены мост и адаптер библиотеки. Пакет может использоваться с фреймворком X, но у вас должна быть возможность с легкостью создать другой пакет-«мост» и заставить его работать и с фреймворком Y. То же самое касается и библиотеки A, для которой адаптер уже доступен. Он реализует интерфейс из пакета, что упрощает предоставление второго адаптера, который заставит пакет работать с библиотекой B.

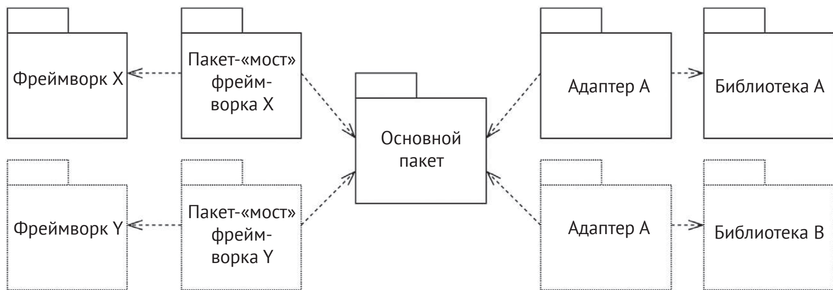


Рис. 5-9. Граф зависимостей после извлечения пакета, когда он становится независимым от фреймворка и библиотек

Не всякий сторонний код требует от вас применения принципа *инверсии зависимостей*. Если вам так и не будет позволено зависеть от какого-либо стороннего кода напрямую, вам придется изобретать все снова и снова. В следующем разделе мы перечислим типы классов, которые определенно нуждаются в интерфейсе. Остаются классы, которые можно использовать как есть. На практике я обнаружил, что это будут классы, которые делают одну вещь, и делают ее хорошо. Не бойтесь зависеть от них, осо-

бенно если специалисты по сопровождению хорошо справляются с проектированием пакетов.

Примеры стороннего кода, от которого вы можете зависеть, можно найти в библиотеках, связанных с:

- утверждениями;
- рефлексией;
- мапированием;
- кодированием/декодированием;
- инфлексией.

Скорее всего, вы сможете добавить еще несколько примеров. На практике вы также можете решить напрямую зависеть от другого конкретного стороннего кода, даже если это потребует инверсии зависимостей. Вы всегда можете добавить интерфейс позже, а сейчас наслаждайтесь созданием поверх готовых строительных блоков, сделанных другими, жертвуя гибкостью ради более раннего выпуска.

Есть еще один вариант, о котором вам следует подумать, когда вы рассматриваете использование стороннего кода. Вместо того чтобы обходить их несколько неуклюжий API или иметь дело с непредсказуемым циклом выпуска, вы также можете скопировать их идею и создать собственную версию. Например, если вам нужен диспетчер событий, но ни один из популярных пакетов диспетчера событий не соответствует вашим ожиданиям, подумайте над тем, чтобы написать его самостоятельно. В любом случае, для этого потребуется не так много кода, и, делая это, вы можете делать его таким, каким хотите. Это не всегда самая эффективная вещь, но, по крайней мере, следует рассмотреть это как вариант. Иногда, когда изобретаешь колесо заново, получается колесо получше.

Когда публиковать явный интерфейс класса

В предыдущих главах мы видели интерфейсы, используемые для того, чтобы сделать классы открытыми для расширения и закрытыми для модификации. Мы узнали, что подклассы могут быть хорошей заменой их интерфейсов. Мы также обсудили, как разделить интерфейсы в соответствии с тем, как они используются. И наконец, мы узнали, как изменить направления зависимости в сторону более абстрактных вещей. Что мы не обсуждали подробно, так это то, какие *классы действительно нуждаются в интерфейсе*. Как уже упоминалось неоднократно, не каждому классу нужен интерфейс. Поэтому, прежде чем мы завершим эту главу и углубимся в принципы проектирования пакетов, давайте сначала ответим на во-

прос: когда следует публиковать явный интерфейс класса? А когда достаточно класса без интерфейса?

Если не все открытые методы предназначены для использования обычными клиентами

Класс всегда имеет *неявный* интерфейс, состоящий из всех его открытых методов. Так, класс будет известен другим классам, которые его используют. Неявный интерфейс можно легко превратить в явный, собрав все эти открытые методы (кроме конструктора, который не должен рассматриваться как обычный метод), вырезав тела методов и скопировав оставшиеся сигнатуры методов в файл интерфейса (*см.* листинг 5-21).

Листинг 5-21. Исходный класс EntityManager и его извлеченный явный интерфейс

```
//Исходный класс только с неявным интерфейсом:
final class EntityManager
{
public function persist(object $object): void
{
// ...
}
public function flush(object $object = null): void
{
// ...
}
public function getConnection(): Connection
{
// ...
}
public function getCache(): Cache
{
// ...
}
// И так далее;
}
// Извлеченный - явный - интерфейс:
interface EntityManagerInterface
{
public function persist(object $object): void;
public function flush(object $object = null): void;
public function getConnection(): Connection;
public function getCache(): Cache;
// ...
}
```

Однако обычным клиентам EntityManager не потребуется доступ к внутренне используемому объекту Connection или Cache, который можно получить, вызвав методы getConnection() или getCache() соответственно. Можно даже сказать, что *неявный* интерфейс класса EntityManager излишне предоставляет доступ к деталям реализации и внутренним структурам данных для клиентов.

Копируя сигнатуры этих методов во вновь созданный интерфейс EntityManagerInterface, мы упустили возможность ограничить размер интерфейса, когда он становится доступным для постоянных клиентов. Было бы очень полезно, если бы клиентам нужно было зависеть только от методов, которые они используют. Таким образом, улучшенный EntityManagerInterface должен оставить только методы persist() и flush(), как показано в листинге 5-22.

Листинг 5-22. Улучшенный интерфейс EntityManagerInterface

```
interface EntityManagerInterface
{
public function persist(object $object);
public function flush(object $object = null);
}
```

Мы обсуждали эту стратегию более подробно в главе 4, когда рассматривали принцип *разделения интерфейса*, который велит вам не позволять клиентам зависеть от методов, которые они не используют (или не должны использовать!).

Если класс использует ввод/вывод

Всякий раз, когда класс выполняет какой-либо вызов, использующий ввод/вывод (сеть, файловая система, источник случайности системы или системные часы), нужно обязательно предоставить интерфейс для него. Причина состоит в том, что в тестовом сценарии вам нужно заменить этот класс на дублера теста, и вам нужен интерфейс для создания этого дублера. Пример класса, который использует ввод/вывод, – CurlHttpClient в листинге 5-23.

Листинг 5-23. Класс CurlHttpClient и его интерфейс

```
// Класс, использующий ввод/вывод:
final class CurlHttpClient
{
public function get(string $url): string
{
$ch = curl_init();
curl_setopt($ch, CURLOPT_URL, $url);
```

```
curl_setopt($ch, CURLOPT_RETURNTRANSFER, true);
// this call uses the network!
$result = curl_exec($ch);
// ...
return $result;
}
}
//Явный интерфейс для таких HTTP-клиентов, как CurlHttpClient;
interface HttpClient
{
public function get(string $url): string;
}
```

Если вы хотите узнать больше об использовании дублеров для замены реальных вызовов ввода-вывода, взгляните на мою серию статей[1].

Если класс зависит от стороннего кода

Если в вашем классе используется какой-либо сторонний код (например, из пакета, который вы сами не обслуживаете), может быть целесообразно изолировать интеграцию вашего кода с этим сторонним кодом и скрыть детали, стоящие за интерфейсом. Подходящие причины для этого:

- (неявный) интерфейс не будет таким, каким вы бы сами его разработали;
- вы не уверены в том, что пакет безопасен.

Допустим, вам нужен инструмент для вычисления различий между двумя мультиломаными. Существует пакет с открытым исходным кодом (`nicky/funky-diff`), который обеспечивает более или менее то, что вам нужно, но API немного не в порядке. Вам нужна строка с плюсами и минусами, но класс из этого пакета возвращает список объектов `ChunkDiff` (*см.* листинг 5-24).

Листинг 5-24. Класс FunkyDiffer

```
class FunkyDiffer
{
/**
* @param array $from Lines
```

[1] Mocking at architectural boundaries: persistence and time // https://matthiasnoback.nl/2018/02/mocking-at-architectural-boundaries-persistence-and-time/; «Mocking at architectural boundaries: the filesystem and randomness // https://matthiasnoback.nl/2018/03/mocking-the-filesystem-and-randomness/; «Mocking the network //https://matthiasnoback.nl/2018/04/mocking-the-network/.

```
* @param array $to Lines to compare to
* @return array|ChunkDiff[]
*/
public function diff(array $from, array $to)
{
// ...
}
}
```

Помимо того что тут предлагается странный API, пакет «обслуживается» человеком, о котором вы никогда и не слышали (и у него 15 открытых вопросов и 7 запросов на принятие изменений). Таким образом, вы должны защитить стабильность своего пакета и определить собственный интерфейс. Затем вы добавляете класс-*адаптер*[1], который реализует ваш интерфейс, но при этом делегирует работу классу `FunkyDiffer`, как показано в листинге 5-25.

Листинг 5-25. Адаптер для FunkyDiffer

```
interface Differ
{
public function generate(string $from, string $to): string;
}
final class DifferUsesFunkyDiffer implements Differ
{
private $funkyDiffer;
public function __construct(FunkyDiffer $funkyDiffer)
{
$this->funkyDiffer = $funkyDiffer;
}
public function generate(string $from, string $to): string
{
return implode(
"\n",
array_map(
function (ChunkDiff $chunkDiff) {
return $chunkDiff->asString();
},
$this->funkyDiffer->diff(
explode("\n", $from),
explode("\n", $to)
)
)
);
}
}
```

[1] *Erich Gamma e. a.* Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

Преимущество этого подхода состоит в том, что теперь вы всегда можете переключиться на другую библиотеку, не меняя объем своего кода. Только класс-адаптер необходимо переписать, чтобы использовать ее.

Кстати, старый добрый шаблон *Фасад*[1] также можно использовать здесь в качестве варианта (*см.* Листинг 5-26), поскольку он будет скрывать использование сторонней реализации. Однако из-за отсутствия явного интерфейса вы не сможете поэкспериментировать с альтернативными реализациями.

То же самое касается пользователей вашего пакета: они не смогут написать собственную реализацию «differ».

Листинг 5-26. Фасад для FunkyDiffer

```
final class Differ
{
public function generate(string $from, string $to): string
{
$funkyDiffer = new FunkyDiffer();
// delegate to FunkyDiffer
}
}
```

Если вы хотите ввести абстракцию для множества конкретных вещей

Если вы хотите рассматривать разные специфические классы неким одинаковым для каждого из них образом, то вы должны использовать интерфейс, который охватывает их общую основу. Такой интерфейс часто называют «абстракцией», потому что он абстрагирует детали, которые не имеют значения для клиента этого интерфейса. Хорошим примером является `VoterInterface` из компонента Symfony Security. Каждое приложение имеет свою собственную логику авторизации, но `AccessDecisionManager`[2] не волнуют точные правила. Он может работать с любым написанным вами избирателем, если он реализует `VoterInterface` и работает в соответствии с инструкциями, приведенными в документации этого интерфейса. Пример такой реализации показан в листинге 5-27.

---

[1] *Erich Gamma e.* a. Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

[2] https://github.com/symfony/security/blob/v4.1.6/Core/Authorization/AccessDecisionManager.php.

Листинг 5-27. Пример реализации VoterInterface

```
final class MySpecificVoter implements VoterInterface
{
public function vote(
TokenInterface $token,
$subject,
array $attributes
): int {
// ...
}
}
```

В случае с `VoterInterface` лица, обеспечивающие поддержку пакета, предлагают пользователям своего пакета способ предоставить собственные правила авторизации. Но иногда абстракция существует только для кода в самом пакете. И в этом случае не стесняйтесь добавлять ее.

Если вы предвидите, что пользователь захочет заменить часть иерархии объектов

В большинстве случаев `final class` – это лучшее, что вы можете создать. Если пользователю не нравится ваш класс, он может просто отказаться от его использования. Однако если вы создаете иерархию объектов, то должны ввести интерфейс для каждого класса. Таким образом, пользователь может заменить определенный фрагмент логики где-то в этой иерархии своей собственной логикой, что сделает ваш код полезным при максимально возможном количестве ситуаций.

Вот пример, взятый с сайта https://tactician.thephpleague.com, где предлагается реализация командной шины.

Пакет поставляется с классом `CommandBus`[1] (*см.* листинг 5-28). Это класс, а не интерфейс, потому что его неявный интерфейс не больше, чем его явный интерфейс – единственный открытый метод – `handle()`.

Листинг 5-28. Класс CommandBus (сокращенный вариант)

```
class CommandBus
{
// ...
public function __construct(array $middleware)
{
```

[1] https://github.com/thephpleague/tactician/blob/v1.0.3/src/CommandBus.php.

```
// ...
}
public function handle($command)
{
// ...
}
// ...
}
```

Чтобы настроить работающий экземпляр CommandBus, вам нужно создать несколько экземпляров классов-посредников, которые реализуют интерфейс Middleware[1] (*см.* листинг 5-29). Ниже приводится пример интерфейса, который был представлен как абстракция, позволяющий сопровождающему пакету обрабатывать несколько конкретных вещей неким общим способом, а также позволять пользователям подключать собственные конкретные реализации.

Листинг 5-29. Интерфейс Middleware (сокращенный вариант)

```
interface Middleware
{
public function execute($command, callable $next);
}
```

Одним из этих интерфейсов-посредников является CommandHandlerMiddleware[2], который сам нуждается в «экстракторе имени команды», «локаторе обработчика» и «инфлекторе имени метода». Все они имеют реализацию по умолчанию внутри пакета (имя команды – *это* имя класса, обработчик команды хранится в памяти, а метод handle – это handle плюс имя команды), как показано в листинге 5-30.

Листинг 5-30. Настройка CommandHandlerMiddleware

```
$handlerMiddleware = new CommandHandlerMiddleware(
new ClassNameExtractor(),
new InMemoryLocator([...]),
new HandleClassNameInflector()
);
$commandBus = new CommandBus(
```

---

1 https://github.com/thephpleague/tactician/blob/v1.0.3/src/Middleware.php.

2 https://github.com/thephpleague/tactician/blob/v1.0.3/src/Handler/CommandHandlerMiddleware.php.

```
[
...,
$handlerMiddleware,
...
]
);
```

Каждый взаимодействующий объект, который вводится в `CommandHandlerMiddleware`, можно легко заменить повторной реализацией интерфейсов этих объектов (`CommandNameExtractor`, `HandlerLocator` и `MethodNameInflector` соответственно).

Поскольку `CommandHandlerMiddleware` зависит от интерфейсов, а не от конкретных классов, он останется полезным для своих пользователей, даже если они захотят заменить часть встроенной логики своей собственной, например когда они хотели бы использовать свой любимый локатор служб, чтобы получить обработчика команд.

Кстати, добавление интерфейса для этих взаимодействующих объектов также помогает пользователю *декорировать* существующие реализации интерфейса, используя композицию объектов.

Для всего остального: придерживайтесь финального класса

Если ваша ситуация не соответствует ни одной из описанных ранее, скорее всего, лучшее, что вы можете сделать, – это *не* добавлять интерфейс, а просто придерживаться использования класса, предпочтительно класса `final`. Преимущество маркировки класса как «final» заключается в том, что подклассы больше не являются официально поддерживаемым способом изменения поведения класса. Это избавит вас от многих неприятностей позже, когда вы смените данный класс в качестве сопровождающего пакета. Вам не придется беспокоиться о пользователях, которые полагаются на внутреннее устройство вашего класса каким-то неожиданным образом.

Классы, которые почти никогда не нуждаются в интерфейсе:

- классы, моделирующие некую концепцию из вашей предметной области (сущности и объекты-значения);
- классы, которые в противном случае представляют объекты с фиксацией состояния (в отличие от классов, представляющих службы без сохранения состояния);
- классы, представляющие определенный фрагмент бизнес-логики или вычисление.

Общим для этих типов классов является то, что вовсе не нужно и не желательно менять их реализации.

Заключение

Как мы видели в этой главе, следовать принципу *инверсии зависимостей* полезно, когда другие начинают использовать ваши классы. Они хотят, чтобы ваши классы были абстрактными, только в зависимости от других абстрактных вещей, оставляя детали для пары небольших классов с конкретными ответственностями.

Применение принципа *инверсии зависимостей* в вашем коде облегчит пользователям замену определенных частей вашего кода другими частями, адаптированными к их конкретной ситуации. В то же время *ваш* код остается общим и абстрактным, и поэтому его можно использовать повторно.

Часть II

Разработка пакетов

Код состоит из *операторов*, сгруппированных в *функции*, сгруппированные в *классы*, сгруппированные в *пакеты*, объединенные в *системы*. Существует несколько идей касательно этой цепочки концепций, которые я хотел бы обсудить здесь, прежде чем мы перейдем к углубленному изучению фактических *принципов разработки пакетов*.

Стать программистом

Мне пришло в голову, что на протяжении своей карьеры программиста я узнавал об этих различных концепциях в том же порядке, в котором только что упомянул их. Первое, что я научился делать в PHP, будучи молодым создателем веб-сайтов, – это вставлять операторы PHP в обычный HTML-файл. Таким образом можно было превратить статическую HTML-страницу в динамическую. Вы можете условно показать некоторые вещи на странице, динамически построить дерево навигации, выполнить некую обработку форм и даже извлечь что-то из базы данных.

```
<?php
$name = htmlentities($_GET['name'], ENT_QUOTES);
$day = date('l');
?>
<html>
<head>
<title>My first homepage</title>
</head>
<body>
<h1>Welcome, <?=$name?></h1>
<p>Today it's <?=$day?></p>
</body>
</html>
```

Когда созданные мной страницы стали более сложными, я почувствовал необходимость улучшить это, облегчить себе работу и поддержать свое будущее я, когда клиент снова передумает. Поначалу я использовал так называемые *файлы* `include`. На поведение этих файлов можно было влиять с помощью глобальных переменных:

```
<?php
// display_day.php
global $day;
?><p>Today it's <?=$day?></p>
<?php
// index.php
global $day;
$day = date('l');
include('display_day.php');
```

Оглядываясь на этот странный код из моих самых старых проектов, я понимаю, что на самом деле использовал эти файлы в качестве неких *функций* (хотя и не по-настоящему функциональным образом, потому что у этих «функций» были некоторые неприятные побочные эффекты, такие как отправка вывода напрямую клиенту).

Использование файлов `include` в качестве функций прекрасно работало некоторое время, пока требования клиентов не стали более сложными, и мне пришлось создать механизм аутентификации для пользователей, используя форму входа в систему. Я скопировал некий код из интернета, в котором содержались *реальные функции*. Конечно, я просто вставил этот код в свой проект (и это сработало). Но потом я начал распутывать его, углубляясь в эту «новую» концепцию функции.

Когда разобрался, все резко изменилось. При каждой возможности ввести новую функцию я добавлял ее в файл `functions.php`. Этот файл был в каждом сценарии, чтобы все эти функции были доступны везде:

```
function wrap($text, $maxLineLength) {
// ...
}
function fetch_user_data($id) {
// ...
}
function array_merge_deep($array1, $array2) {
// ...
}
function copy_shopping_cart() {
// ...
```

```
}
function rename_file($source, $destination) {
// ...
}
// ...
```

Несмотря на то что многие из этих функций по-прежнему отражали результаты непосредственно в клиенте (вместо буферизации выходных данных), я чувствовал, что мои приложения уже становятся довольно продвинутыми (конечно, я копировал этот файл `functions.php` в каждый новый проект, который начинал).

В какой-то момент я просматривал сайт https://secure.php.net и наткнулся на страницу, посвященную классам[1]. Я понял, что это способ группировки функций, которые были связаны друг с другом. Поэтому создал большой класс `Page`, ставший ядром первой CMS, которую я когда-либо создавал. Я добавил исходный код класса `Page` в качестве приложения к этой книге для вашего удовольствия, но позвольте мне показать вам некоторые наиболее интересные фрагменты кода здесь.

```
class Page
{
public $uri = null;
public $page = array();
public $site_title = «;
public $breadcrumbs = array();
public $js = array();
public $css = array();
public $auto_include_dir = «;
/* @public $smart Smarty */
public $smarty = null;
public $default_template = «;
public $template = «;
public $cms_login = null;
public $user_login = null;
public $is_user = false;
public $is_admin = false;
public $languages = array();
public $default_language = null;
public $language = null;
public $menu_items = array();
protected $_extra_request_parameters = array();
public function __construct($uri)
```

[1] https://secure.php.net/manual/en/language.oop5.basic.php.

```
{
$this->connect_db();
header('Content-Type: '.HEADER_CONTENT_TYPE);
$this->smarty = new Smarty;
if (isset($_GET['clear_cache']))
{
$this->smarty->clear_cache();
}
if (DEBUGGING)
{
$this->smarty->caching = false;
if (trusted_ip())
{
$this->smarty->debugging = true;
}
}
if (trusted_ip())
{
ini_set('display_errors', '1');
error_reporting(
109
E_ERROR | E_PARSE | E_WARNING | E_USER_ERROR
| E_USER_NOTICE | E_USER_WARNING
);
}
else
{
$this->smarty->debugging = false;
ini_set('display_errors', '0');
error_reporting(0);
}
// ...
if (!table_exists('content'))
{
require(ROOT.'/includes/install.php');
install();
}
$this->add_title_part(SITE_TITLE);
$this->cms_login = new LoginClass('admins', 'cms_login');
$this->user_login = new LoginClass('users', 'user_login');
if ($this->cms_login->isLoggedIn())
{
$this->is_admin = true;
}
if ($this->user_login->isLoggedIn())
{
$this->is_user = true;
}
```

```
// ...
$this->open_page();
}
public function connect_db()
{
$this->db_connection = @mysql_connect(
MYSQL_HOST,
MYSQL_USER,
MYSQL_PASSWORD
);
if ($this->db_connection)
{
$this->db = @mysql_select_db(MYSQL_DB);
if (!$this->db)
{
?><p class="warning">Geen database!</p><?
exit;
}
}
else
{
?><p class="warning">Geen verbinding!</p><?
exit;
}
}
}
```

Вот несколько основных моментов из этого шедевра:

- класс `Page` содержит около 20 или 30 переменных экземпляра;
- он зависит примерно от 20 констант, которые определены вне этого класса (более конкретно: в файле `config.php`);
- он относится к так называемым суперглобальным переменным, таким как `$_GET`;
- он глобально изменяет настройки PHP-процесса путем настройки режима обработки ошибок в конструкторе;
- сообщения об ошибках (на нидерландском языке) выводятся непосредственно пользователю, после чего программа просто завершает работу;
- он подключается к базе данных MySQL и проверяет наличие необходимых таблиц. Если их нет, он запускает скрипт установки;
- он отправляет заголовки ответа.

Хотя сегодня я бы ни за что не написал такой код, когда я сейчас смотрю на класс `Page`, мне не стыдно за то, что я делал тогда. Мне

ясно, что я прилагал недостаточно усилий, чтобы все *функционировало* (потому что все просто работало; я всегда старался, чтобы работа была выполнена хорошо). Скорее, я изо всех сил пытался *навести порядок*.

Самое трудное

Мне потребовалось несколько лет, прежде чем я научился делать свои классы *умеренно хорошими*. И все же каждый день появляется что-то новое, что ты узнаешь при проектировании классов, какая-то старая привычка, от которой следует отказаться, какой-то новый принцип, который нужно применить. Из этого я делаю вывод, что организация кода в классы – *сложная вещь*. Довольно легко выучить ключевые слова, которые предоставляет язык программирования для работы с классами (`class`, `extends`, `implements`, `abstract`, `final` и т. д.). Гораздо сложнее научиться правильно их использовать.

Давайте вернемся к этой цепочке понятий: операторы, сгруппированные в функции, сгруппированные в классы, сгруппированные в пакеты, объединенные в систему. Позвольте мне спросить: в чем «суть» программы? Это *операторы*. Операторы приводят все в действие. Если бы нам нужно было преобразовать все методы класса программы в обычные функции, а затем мы бы встраивали эти функции, то получили бы одну длинную страницу, состоящую из операторов, и при их выполнении программа все равно делала бы то же самое.

Исходя из этого, можно сделать вывод, что код не *должен* быть упорядочен, если вам нужно всего лишь заставить его *просто работать*. Для компьютера важны операторы. Тем не менее мы прилагаем огромные усилия для их *модулирования*. Мы помещаем их в методы классов и группируем классы в пакеты. И, судя по порядку, согласно которому вы обучаетесь, будучи разработчиком, написание классов и создание пакетов гораздо сложнее, чем простое написание операторов.

Принципы связности

На протяжении многих лет вы пытаетесь упорядочить свой код наилучшим образом. При этом вы постепенно развиваете сильное

чувство «принадлежности друг к другу». Это поможет вам решить, должны ли две части кода быть связаны друг с другом. Ваша интуиция постоянно развивается: когда вы пишете операторы и помещаете их в функции, когда вы пишете классы для этих функций и когда объединяете эти классы в пакеты.

Эта интуиция у вас как у программиста, это чувство «связи друг с другом» на самом деле относится к понятию, именуемому *связностью*. Связность – это *мера силы взаимосвязанности*. Некоторые вещи сильно связаны, а некоторые в меньшей степени, в зависимости от того, насколько они соотносятся друг с другом.

В начале жизненного пути вы учитесь определять, являются ли вещи связанными. В школе вы получаете следующие небольшие упражнения: «Одно из приведенных ниже слов не входит в список, какое? Утка, лягушка, рыба, верблюд». Когда у вас есть список, состоящий из слов «утка, лягушка, рыба», у вас есть очень связанный список слов. Слова означают понятия, которые тесно связаны друг с другом (потому что все это названия животных, которые живут или выживают в воде). Когда вы добавляете в этот список слово «верблюд», он определенно становится менее связанным. Это похоже на работу программиста: вам нужно выяснить, что можно добавить, не делая все это *менее связанным*, и что можно удалить, чтобы сделать все это *более связанным*.

В контексте проектирования пакетов связность главным образом касается того, какие классы связаны друг с другом в пакете. Есть много разных способов, с помощью которых можно упорядочить и комбинировать классы, и все они дают разные виды связности. Например, можно сгруппировать все классы, которые служат в качестве контроллера, или все классы, которые являются сущностями. В результате появляется то, что называется *логической связностью*. Но когда вы группируете контроллер «запись в блоге» и сущность «запись в блоге», получается *коммуникационная связность*: все классы в таком пакете оперируют одними и теми же данными, то есть записями в блоге из базы данных.

Существует несколько других типов связности (последовательная, временная и т. д.), но наиболее важным типом связности, к которому следует стремиться, является *связность функциональная*. Функциональная связность достигается, когда все, что находится в «модуле» (например, в пакете), можно использовать для выполнения одной, четко определенной задачи.

Принципы проектирования классов способствуют связности

Ранее я уже говорил, что чувство связи друг с другом, возникающее у программиста, основано на интуиции, сформированной опытом. Конечно, есть и рациональная сторона этого ощущения: если вы знаете принципы проектирования классов, это поможет вам написать высоко связанный код. Например, когда вы применяете принципы SOLID к своим классам, они автоматически становятся более связанными:

- в итоге у вас будут классы с узкоспециализированными интерфейсами, предназначенными для конкретного клиента. В результате тот факт, что эти интерфейсы содержат методы, которым там не место, будет маловероятен. Таким образом, применяя принцип *разделения интерфейса*, вы получите высоко связанные классы;
- у вас также будут классы всего лишь с одной причиной для изменений, что означает, что они не разбросаны «повсюду». Таким образом, применение принципа *единственной ответственности* также имеет положительный эффект, когда классы становятся более связанными.

Как только мы узнаем, как создавать высоко связанные классы, можно сделать следующий шаг. После операторов, функций и классов мы переходим к пакетам. Пакеты – это группы классов, и, как и все, что является группой, состоящей из разных элементов, пакет также обладает связностью (определенной степенью взаимосвязанности). Если классы были сгруппированы произвольно, пакет, содержащий их, обладал бы *случайной связностью*. Конечно, хитрость заключается в том, чтобы сгруппировать классы таким образом, чтобы пакет имел высокий уровень функциональной связности: все классы в пакете должны служить выполнению одной, четко определенной задачи.

Существует три принципа проектирования пакетов, которые помогут вам в создании высоко связанных пакетов. В последующих главах мы подробно обсудим каждый из них. Они называются принципом *эквивалентности повторного использования и выпуска* (глава 6), принципом *совместного повторного использования* (глава 7) и принципом *общей закрытости* (глава 8) соответственно. Применение этих принципов позволит получить менее крупные пакеты, которые легче сопровождать и использовать.

Важно сосредоточиться на пакете и степени его связности, но не менее важно рассмотреть, как пакет ведет себя по отношению к другим пакетам, иными словами, насколько они взаимозависимы.

Принципы зацепления

Большинство классов не могут выжить сами по себе: у них есть некая зависимость, а скорее всего, даже несколько зависимостей. Возможно, им нужен экземпляр другого класса, чтобы делегировать часть работы. Или они *производят* экземпляры иного класса. То есть, многие классы зависят от иных классов, что фактически связывает их друг с другом.

Как мы видели в первой части этой книги, применение принципов SOLID при проектировании классов оказывает благотворное влияние на взаимосвязь между классами. Как вы, возможно, помните, согласно принципу *инверсии зависимостей* класс должен зависеть только от другого класса или интерфейса, который является абстрактным, а не конкретным. Он также не должен зависеть от классов более низкого уровня, лишь от классов верхнего уровня. И согласно принципу *открытости/закрытости* класс должен быть открыт для расширения, но закрыт для модификации. Это означает, что его поведение должно быть изменяемым, без фактического изменения его кода.

Когда мы обсуждали принцип инверсии зависимостей, мы уже вкратце рассматривали ситуацию, когда один класс зависит от класса в другом пакете. Класс из одного пакета, который зависит от класса из другого пакета, вводит новый уровень зацепления, именуемый *зацеплением пакетов*.

Если вы работали с пакетами и менеджером зависимостей или пакетов, то уже знаете, что зацепление пакетов может пойти не так во многих отношениях. Часто возникают проблемы из-за несовместимых версий зависимостей. Или каким-то образом происходят круговые зависимости. Возможно, нестабильные пакеты, которые могут изменяться, часто приводят к выходу из строя вашего собственного проекта. Или, может быть, некоторые из ваших зависимостей сами имеют нестабильные зависимости, последствия которых распространяются на ваш собственный код.

При возникновении такого рода проблем нам необходимы некие руководящие принципы, которые помогают разрабатывать пакеты с хорошими зависимостями. Нам нужны пакеты, которые

могут быть надежными зависимостями других пакетов и проектов. Соответствующие принципы разработки пакетов называются «принципами зацепления», которые мы обсудим в последних трех главах: принцип ацикличности зависимостей (глава 9), принцип устойчивых зависимостей (глава 10) и принцип устойчивых абстракций (глава 11), – после того как полностью рассмотрим принципы связности.

Глава 6

Принцип эквивалентности повторного использования и выпуска

Первым из фактических принципов разработки пакетов, обсуждаемых в этой книге, является принцип эквивалентности повторного использования и выпуска. Данный принцип гласит[1]:

> *Единица повторного использования равна единице выпуска.*

У этого принципа две стороны. Прежде всего вы должны написать столько кода, сколько вы (или другие) можете обоснованно использовать повторно. Нет смысла тратить все время и силы, необходимые для правильного выпуска кода, если никто не будет использовать его в другом проекте. Для этого может потребоваться провести исследование, чтобы установить жизнеспособность вашего пакета, как только вы выпустите его в частном или открытом порядке. Возможно, вам только кажется, что этот пакет предназначен для повторного использования, но в конце концов он окажется полезным лишь в вашем конкретном случае использования.

Другая сторона принципа заключается в том, что вы можете повторно использовать только тот объем кода, который *действительно можете выпустить*. Применяя все принципы проектирования классов, вы, возможно, создали идеально обобщенный, многократно используемый код. Но если вы никогда не выпустите его, его нельзя использовать повторно. Поэтому,

---

[1] *Robert C. Martin*. The principles of OOD // http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod .

прежде чем приступить к тому, чтобы сделать весь код пригодным для повторного использования, попробуйте сначала ответить на вопрос: сможете ли вы выпустить этот код и управлять будущими выпусками?

Знание усилий, необходимых для выпуска пакета, поможет вам определиться с количеством и размером пакетов, которые вы собираетесь создать. Выпуск сотни маленьких пакетов – это то, что сделать невозможно. Каждый пакет требует определенного количества времени и сил от того, кто отвечает за поддержку. Подумайте об отслеживании и устранении проблем, добавлении тегов версий в новые выпуски, обновлении документации и т. д. С другой стороны, выпуск одного очень большого пакета также невозможен. Он претерпит такое количество изменений, связанных с различными частями пакета, что станет очень изменчивым, что совсем не полезно для его пользователей.

Как я объяснил во введении, связность – это мера силы взаимосвязанности. И поэтому принципы связности разработки пакетов предлагают стратегии, позволяющие решить, следует ли группировать классы в пакете. Принцип *эквивалентности повторного использования и выпуска* помогает вам решить, сможете ли вы выпустить такой пакет вообще. Он заставляет вас осознать тот факт, что выпускаемый пакет требует заботы со стороны лица, отвечающего за его поддержку.

Остальные разделы этой главы представят обзор всего того, о чем нужно позаботиться, когда вы приступите к выпуску пакетов. В то время как предыдущая часть книги была о том, как подготовить свои *классы* к повторному использованию, в этой главе говорится о том, как подготовить свой *пакет* классов для повторного использования, то есть для выпуска.

Хотя мы и будем говорить о коде и обсудим тактику обеспечения его обратной совместимости, вы должны знать, что вторая половина этой главы посвящена другим, более практичным темам, связанным с разработкой пакетов, таким как семантическое управление версиями, обеспечение качества и метафайлы, которые должны присутствовать в пакете. Если вас (пока) не интересуют эти темы, потому что вы уже знаете о них или хотите подробно изучить их только тогда, когда вы действительно приступаете к выпуску пакета, можете пропустить следующие разделы и сразу перейти к заключению.

Держите свой пакет в системе управления версиями

Первое, что нужно сделать, – это настроить систему управления версиями для своего пакета. Вы должны иметь возможность отслеживать изменения, внесенные вами или кем-либо из участников, а люди должны иметь возможность получать последнюю версию пакета. Таким образом, даже если рассылка фрагментов кода технически может являться своего рода контролем версий (дата отправки сообщения может использоваться в качестве номера версии пакета), всегда нужно использовать реальную систему управления версиями (такую как Git[1]).

Если у вас есть идея для создания пакета (который в первую очередь представляет собой согласованный набор классов), первое, что вы делаете, – это настраиваете для него репозиторий. Это позволит вам вернуться к предыдущим ситуациям, если одно конкретное изменение поставило под угрозу весь проект. Если вы работаете в команде, использование системы управления версиями также помогает предотвратить конфликтующие изменения. Она позволяет вам работать с новой или экспериментальной функцией и тестировать ее в отдельной *ветке*, не ставя под угрозу стабильность главной ветки пакета.

Репозиторий системы управления версиями должен рассматриваться как полное описание *истории* проекта. Вы и ваша команда будете использовать репозиторий как способ выяснить, когда или почему появилась ошибка. Убедитесь, что вы фиксируете в репозитории только те изменения, которые являются связанными, и добавляйте пояснительные комментарии, когда фиксируете что-то.

Чтобы сделать пакет доступным, нужно убедиться, что он где-то размещен. В зависимости от ваших потребностей это может быть что-то общедоступное или частное, он может быть размещен на чужом сервере или на своем.

Добавьте файл определений

Большинство языков программирования имеют стандартизированный способ определения пакетов. И часто это простой файл,

---

[1] https://git-scm.com.

который предоставляет некоторые или все приведенные ниже свойства пакета:

- название пакета;
- лица, отвечающие за его поддержку, возможно, некоторые участники (те, кто вносит свой вклад в проект);
- URL-адрес и тип репозитория системы управления версиями;
- необходимые зависимости, как, например, другие пакеты, определенные языковые версии и т. д.

Читайте как можно больше о различных вариантах. Пакет, содержащий развернутый файл определений, который правильно использует все параметры, вероятно, будет хорошо себя вести в экосистеме пакетов вашего языка программирования.

После того как вы создали правильный файл определений пакета, вам, вероятно, придется зарегистрировать пакет в каком-то центральном хранилище пакетов или реестре. У каждого языка программирования имеются свои собственные удаленные репозитории пакетов, с различными руководствами и требованиями.

Семантическое версионирование

Когда вы выпускаете пакет, то должны ответить на следующие вопросы и прояснить свои намерения:

- вносите ли вы изменения в API вашего кода с большой осторожностью?
- будете ли вы пытаться убедиться, что эти изменения не нарушат способ, с помощью которого пользователи взаимодействуют с вашим пакетом?
- в каких ситуациях вы бы позволили себе в значительной степени изменить свой API?

Другими словами, как вы собираетесь позаботиться об *обратной совместимости*? Лица, отвечающие за поддержку пакетов, обычно следуют стратегии управления версиями, именуемой «семантическим версионированием». Схема этой стратегии выглядит так:

- исправьте ошибки и выпустите их как версии *патча* (например, x.x.1 ⇒ x.x.2);
- добавьте в свой пакет что-нибудь новое, но не забудьте выпускать новую *младшую* версию каждый раз, когда вы делаете это (например, x.1.x ⇒ x.2.x). Если вы хотите исключить какие-то вещи, попридержите их ненадолго;

- удалите устаревшие фрагменты или используйте обратно несовместимые изменения при выпуске новой *старшей* версии (например, 1.x.x ⇒ 2.x.x).

Первая часть номера версии пакета, номер перед первой точкой, называется *старшей версией*. Обычно первая старшая версия – это 0. Эта версия должна считаться незаконченной, экспериментальной, сильно изменяющейся без особой заботы об обратной совместимости. Начиная со старшей версии 1 общедоступный API должен стать стабильным, и с этого момента пакет обладает определенной надежностью. Каждое последующее увеличение номера старшей версии отмечает момент, когда часть кода нарушает обратную совместимость. Это момент, когда сигнатуры методов меняются, а устаревшие классы или интерфейсы удаляются. Иногда даже полная переделка одного и того же функционала выпускается в виде новой *старшей* версии.

Вторая часть номера версии – *младшая версия*. Она также начинается с 0, хотя это не имеет особого значения, кроме как «быть первым». Младшие версии могут увеличиваться, когда в пакет добавляются новые функции или когда части существующего общедоступного API помечаются как устаревшие. Новая младшая версия обещает, что для ее пользователей ничего не изменится, существующие способы использования пакета не будут нарушены. Младшая версия лишь добавляет новые способы использования пакета.

Последняя часть номера версии – это *версия патча*. Начиная с версии 0 она увеличивается с появлением каждого патча, выпущенного для пакета. Это может быть либо исправление ошибки, либо некий закрытый код, подвергшийся рефакторингу, то есть код, который недоступен только с помощью общедоступного API пакета. Поскольку рефакторинг означает изменение структуры кода без изменения его поведения, рефакторинг закрытого кода пакета не будет иметь никаких негативных побочных эффектов для существующих пользователей.

Сразу после номера версии (состоящей из старшей, младшей версий и версии патча, разделенных точками) может идти текстовое указание состояния пакета: `alpha`, `beta`, `rc` (версия-кандидат), за которыми может следовать еще одна точка и еще одно число.

Число в сочетании с необязательным метаидентификатором можно использовать для сравнения номеров версий. В листинге 6-1 показан отсортированный список номеров версий.

Листинг 6-1. Отсортированные номера версий

```
1.9.10
2.0.0
2.1.0
2.1.1-alpha
2.1.1-beta
2.1.1-rc.1
2.1.1-rc.2
2.1.1
```

Сравнение выполняется естественным путем, поэтому версия 2.1.1 является более низкой версией, чем версия 2.10.1. Ни для какой из частей номера версии нет ограничений, поэтому вы можете просто увеличивать его и дальше.

Проектирование для обратной совместимости

Когда вы используете семантическое управление версиями для своих пакетов, обеспечение обратной совместимости означает, что вы стремитесь предоставить точно такую же функциональность в младшей версии x + 1, что и в предыдущей версии x. Другими словами, если какой-то пользовательский код опирается на функцию, предоставленную версией 1.1.0, вы гарантируете, что та же самая функция будет доступна и в версии 1.2.0. Его использование будет иметь одинаковый эффект в обеих версиях. Мало того, что оно будет иметь те же поведенческие эффекты, это свойство также можно будет вызвать таким же образом.

Конечно, вы могли исправить некоторые ошибки в промежутке между двумя младшими версиями, и, возможно, вы *добавили* некоторые функции. Но ничего из этого не должно создавать проблем для пользователей, которые обновляют свою зависимость от вашего пакета до следующей младшей версии. Все их тесты должны пройти, и все должно работать так же, как и до обновления.

Как вы можете себе представить, обеспечить *подлинную* обратную совместимость может быть очень непросто. Вы хотите добиться прогресса, но ваше обещание обратной совместимости может вас сдерживать. Тем не менее если вы хотите, чтобы ваш пакет использовался другими разработчиками, вам нужно предоставить им и новые свойства, *и* непрерывность.

Бывает, что вам не *нужно* обеспечивать непрерывность. Это происходит, когда младшей версией вашего пакета по-прежнему остается 0.x.x. В течение этого периода ваш пакет будет считаться

нестабильным в любом случае, и вы можете перемещать все вокруг. Это может раздражать некоторых первых пользователей, но поскольку они знают о том, что пакет все еще нестабилен, они не могут особо жаловаться.

Постоянная работа с версиями пакета 0. *, похоже, избавит вас от боли, связанной с сохранением обратной совместимости. Однако *нестабильный* пакет, скорее всего, не будет использоваться ни в одном серьезном проекте, который сам по себе намерен быть *стабильным*. Такой проект вызовет раздражение пользователей, когда они будут обновлять его, а ничего больше не работает. Им придется добавлять дополнительные интеграционные тесты для проверки границы между их кодом и вашим, чтобы заметить все проблемы, связанные с совместимостью, на ранних этапах. Они будут бояться обновлять ваш пакет и, следовательно, пропустят все соответствующие исправления ошибок или уязвимостей безопасности.

В заключение вы должны принять решение о разработке своего кода, и как только вы протестируете свой пакет в одном или двух проектах, выпустите его как версию 1.0.0. Если такой вариант вам не подходит, то можете перейти к стратегии, которая будет заключаться в том, чтобы начать работу над версией 2.0.0 и объявить, что вы вскоре прекратите разработку функций для версии 1.0. Используя ветки в системе управления версиями, вы все равно сможете предоставить исправления для предыдущей старшей версии, если захотите.

Практические правила

Несмотря на то что вы можете избежать выпусков только старших версий, более вероятен сценарий, при котором вы также выпустите и младшие версии. Поэтому разработка обратной совместимости должна стать частью вашей стратегии с момента выпуска первой старшей версии вашего пакета. В следующих разделах я обсуждаю некоторые вещи, которые нужно и не нужно делать, для того чтобы обеспечить обратную совместимость.

Это всего лишь примеры и практические правила. Есть еще много способов, с помощью которых можно предотвратить нарушение обратной совместимости, и все же они могут случайно произойти. Программное обеспечение — по своей природе уже вещь сложная, но есть также способы, с помощью которых люди используют ваш код, который вы официально не поддерживаете или о котором пока не знаете. Это означает, что вам так и не удастся уследить за всем. Но вы можете, по крайней мере, минимизировать потенциальный ущерб.

Ничего не выбрасывайте

Всякий раз, когда вы добавляете что-то в свой пакет, убедитесь, что оно остается в следующей версии. Это относится к таким вещам, как:

- классы;
- методы;
- функции;
- параметры;
- константы.

Класс *существует*, если может быть загружен автоматически, поэтому необязательно, чтобы классы были в одном и том же файле. Просто убедитесь, что загрузчик классов всегда может их найти. Это означает, что вы можете переместить класс в другой пакет и добавить этот пакет в качестве зависимости.

Когда вы что-то переименовываете, добавьте посредника

Переименование классов возможно, но убедитесь, что старый класс по-прежнему можно инстанцировать (*см.* листинг 6-2).

Листинг 6-2. Старый класс по-прежнему доступен

```
/**
* @deprecated Use NewClass instead
*/
class DeprecatedClass extends NewClass
{
// Унаследует все методы из NewClass;
}
class NewClass
{
// ...
}
```

Для этого может потребоваться временно удалить ключевое слово `final` из объявления старого класса или использовать композицию объектов в качестве альтернативного подхода к сохранению совместимости.

Переименование метода возможно, но убедитесь, что вы перенаправили вызов новому методу (*см.* листинг 6-3).

Листинг 6-3. Старый метод по-прежнему доступен

```
class SomeClass
{
```

```
/**
* @deprecated Используйте newMethod();
*/
public function deprecatedMethod()
{
return $this->newMethod();
}
public function newMethod()
{
// ...
}
}
```

Или, если вы переместили функциональность в другой класс, убедитесь, что он по-прежнему работает, когда кто-то использует старый метод (*см*. листинг 6-4).

Листинг 6-4. Старый метод – посредник для нового

```
class SomeClass
{
/**
* @deprecated Используйте Something::doComplicated();
*/
public function doSomethingComplicated()
{
$something = new Something();
return $something->doComplicated();
}
 }
```

ДОБАВЛЕНИЕ АННОТАЦИИ @DEPRECATED

Всякий раз, когда вы исключаете элемент своего кода, будь то класс, метод, функция или свойство, не нужно немедленно удалять его. Придержите его до выпуска следующей старшей версии. А до тех пор убедитесь, что у него есть аннотация @deprecated. Не забудьте добавить небольшое объяснение и рассказать пользователям, что они должны делать, или как они могут изменить собственный код, чтобы подготовить его к следующей старшей версии, в которой устаревшие вещи будут удалены.

Переименование параметров метода не является проблемой (по крайней мере, в PHP), если их порядок и тип не меняются. Переименование параметров метода, определенного в интерфейсе,

также не является проблемой. Классы, реализующие интерфейс, всегда могут использовать разные имена, при условии что типы параметров одинаковы (см. листинг 6-5).

Листинг 6-5. Переименование параметров прекрасно работает, пока их типы одинаковы

```
// Интерфейс, определенный внутри пакета;
interface SomeInterface
{
public function doSomething(ObjectManager $objectManager);
}
// Класс, созданный пользователем пакета;
class SomeClass implements SomeInterface
{
public function doSomething(ObjectManager $entityManager)
{
// ...
}
}
```

Добавляйте параметры только в конце и со значением по умолчанию

Когда вам нужно добавить параметр в метод, убедитесь, что вы добавляете его в конце существующего списка параметров. Также убедитесь, что новый параметр имеет правильное значение по умолчанию (*см.* листинг 6-6).

Листинг 6-6. Добавляйте новые параметры со значением по умолчанию только в конец сигнатуры метода

```
// Текущая версия;
class StorageHandler
{
public function persist(object $object): void
{
$this->entityManager->persist($object);
/*
* В текущей реализации всегда применяется метод flush();
*/
$this->entityManager->flush();
}
}
// Следующая версия;
class StorageHandler
{
```

```
public function persist(
object $object,
$andFlush = true
): void {
$this->entityManager->persist($object);
// В новой реализации метод flush() используется только по запросу;
if ($andFlush) {
$this->entityManager->flush();
}
}
}
```

Дополнительный параметр `$andFlush` был введен со значением по умолчанию `true`, чтобы убедиться, что новый метод ведет себя точно так же, как и старый.

Методы не должны иметь скрытых побочных эффектов

Не ждите, что пользователи вашего кода будут полагаться на определенный скрытый побочный эффект вызова метода. При последующем изменении кода побочный эффект может исчезнуть, что нарушит код пользователя. *См.* листинг 6-7, где приводится пример такого эффекта.

Листинг 6-7. Пример побочного эффекта, который исчезает в более поздней версии метода

```
// Предыдущая версия;
class Stream
{
public function open(string $file): void
{
/*
* При необходимости в предыдущей реализации создается каталог;
*/
$this->createDirectoryIfNotExists($file);
$this->handle = fopen($file, 'w');
}
private function createDirectoryIfNotExists(
string $file
): void {
// ...
}
}
// Следующая версия;
class Stream
```

```
{
public function open(string $file): void
{
/*
* В новой реализации каталог не создается автоматически;
*/
$this->handle = fopen($file, 'w');
}
}
```

В предыдущей версии, когда мы использовали `Stream::open()`, неявно создавался каталог при открытии файла. Такое поведение оказалось проблематичным в некоторых ситуациях, поэтому следующая версия `Stream ::open()` предоставляет пользователю возможность убедиться в том, что каталог существует. Пользователи могут использовать `Filesystem::isDirectory()` и `Filesystem::createDirectory()` (*см.* листинг 6-8).

Листинг 6-8. Класс Filesystem

```
class Filesystem
{
public function createDirectory(string $directory): void
{
// ...
}
public function isDirectory(string $directory): bool
{
// ...
}
}
```

Конечно, это изменение вызывает нарушение обратной совместимости. Но это можно было бы предотвратить в самом начале; убедитесь, что ни у какого метода нет скрытых побочных эффектов и что каждый из них выполняет что-то одно и только одно. Четко определите это что-то в документации метода.

Версии зависимостей не должны быть очень строгими

Если у вашего пакета есть зависимости, убедитесь, что вы не накладываете слишком много ограничений на их номера версий. Например, когда вы пишете код для нового пакета, вы, возможно, предпочтете работать с последней версией одной из его зависимостей; допустим, это версия 2.4.3.

Если человек, отвечающий за поддержку этого пакета, должным образом позаботился об обратной совместимости, вероятно, ваш пакет будет хорошо работать с версией 2.3, а может, даже и с 2.2 или 2.1. Однако, требуя версию зависимости 2.4.3 или выше, вы фактически исключаете всех пользователей, у которых в проекте установлены более низкие версии этой же зависимости, даже если ваш пакет будет хорошо работать с более старыми версиями.

Существует два решения: заставить ваших пользователей перейти на новую версию этой зависимости или сделать свои требования менее строгими. Поскольку первый вариант может что-то повредить в их проекте (причинить неприятности, причиной которых вы не хотите быть), почти всегда лучше выбрать второй вариант – сделать ваш код совместимым со старыми версиями и ослабить собственные требования.

Это также справедливо и в обратную сторону: если доступна новая стабильная версия пакета. Как лицо, отвечающее за поддержку, вы должны заставить свой пакет работать и с этой новой версией. Вы должны проверить, возможно, так уже и есть, установив новую версию зависимости и выполнив тесты своего пакета. При необходимости внесите изменения в свой код, пока все тесты не будут пройдены. Конечно, вам нужно убедиться, что пакет продолжает работать с предыдущей версией зависимости. Вы можете настроить непрерывный процесс интеграции (мы вернемся к этому), чтобы это было автоматически.

Используйте объекты вместо значений примитивного типа

Для обеспечения обратной совместимости рекомендуется использовать объекты, для которых обычно используются массивы или значения примитивного типа. Рассмотрим, как меняется интерфейс `HttpClientInterface`, по мере увеличения номера версии, как показано в листинге 6-9.

Листинг 6-9. Изменения HttpClientInterface

```
// Версия 1.0.0;
interface HttpClientInterface
{
public function connect(string $host): void;
}
// Версия 1.1.0
```

```
interface HttpClientInterface
{
public function connect(
string $hostname,
int $port = 80
): void;
}
// Версия 1.2.0
interface HttpClientInterface
{
public function connect(
string $hostname,
int $port = 80,
bool $useTls = false
): void;
}
// Версия 1.3.0
interface HttpClientInterface
{
public function connect(
string $hostname,
int $port = 80,
bool $useTls = false,
bool $verifyPeer = true
): void;
}
```

Вместо того чтобы добавлять все больше и больше параметров со значениями по умолчанию, было бы намного проще, если бы у метода `connect()` был один параметр с самого начала, который можно было бы расширить без нарушения обратной совместимости. Мы могли бы объединить все эти отдельные значения (имя хоста, порт и т. д.) в один объект, который легко обновить, с именем `ConnectionConfiguration` (*см.* листинг 6-10).

Листинг 6-10. HttpClientInterface принимает один объект

```
interface HttpClientInterface
{
public function connect(
ConnectionConfiguration $configuration
): void;
}
```

Таким образом, при последующих обновлениях пакета не нужно изменять сигнатуру метода `connect()`, а нужно только добавить новые настройки в класс `ConnectionConfiguration` (*см.* листинг 6-11).

Листинг 6-11. Изменения в ConnectionConfiguration

```
// Версия 1.0.0;
class ConnectionConfiguration
{
private $hostname;
public function __construct(string $hostname)
{
$this->hostname = $hostname;
}
public function getHostname(): string
{
return $this->hostname;
}
}
// Версия 1.0.0;
class ConnectionConfiguration
{
// ...
private $port = 80;
public function getPort(): int
{
return $this->port;
}
public function setPort(int $port): void
{
$this->port = $port;
}
}
// Версия 1.0.0;
class ConnectionConfiguration
{
// ...
private $useTls = false;
public function shouldUseTls(): bool
{
return $this->useTls;
}
public function useTls(bool $useTls): void
{
$this->useTls = $useTls;
}
}
// Версия 1.3.0;
class ConnectionConfiguration
{
// ...
private $verifyPeer = false;
```

```
public function shouldVerifyPeer(): bool
{
return $this->verifyPeer;
}
public function verifyPeer(bool $verifyPeer): void
{
$this->verifyPeer = $verifyPeer;
}
}
```

Пользователи этого пакета не будут сталкиваться с проблемами, когда лицо, отвечающее за поддержку, добавляет дополнительный параметр в класс `ConnectionConfiguration`, если имеется правильное значение по умолчанию.

Используйте объекты для инкапсуляции состояния и поведения

Использование объекта вместо значения примитивного типа, как мы делали в примере с `ConnectionConfiguration`, полезно не только тогда, когда вы стремитесь внести обратно совместимые изменения в сигнатуру метода. Естественная инкапсуляция деталей реализации объекта также помогает поддерживать обратную совместимость.

Возьмем для примера конструктор `ConnectionConfiguration`. В какой-то момент времени было невозможно отдельно настроить порт – вместо этого он был извлечен из имени хоста. В более поздней версии класса был добавлен метод `setPort()`, но для обеспечения обратной совместимости старая логика извлечения порта из имени хоста по-прежнему присутствует (*см.* листинг 6-12).

Листинг 6-12. Метод setHostname() поддерживает некое старое поведение

```
class ConnectionConfiguration
{
private $hostname;
private $port = 80;
public function setHostname(string $hostname): void
{
// Для обеспечения обратной совместимости извлеките порт из имени хоста,
если это применимо:
if (strpos($hostname, ':') !== false) {
list($hostname, $port) = explode($hostname);
$this->setPort($port);
}
$this->hostname = $host;
}
```

```
public function setPort(int $port): void
{
if ($port <= 0) {
throw new InvalidArgumentException(
Port should be larger than 0
);
}
$this->port = $port;
}
// ...
}
```

Поддержка различных типов аргументов также может быть отличным способом нормализации данных и обеспечения обратной совместимости с течением времени. Например, вы можете обновить тип параметра с одного значения до списка значений, опуская тип (или перегружая метод, если ваш язык поддерживает это), без нарушения обратной совместимости. Или можно обновить значение примитивного типа до объекта, если клиент по-прежнему использует такое значение, но внутри уже используется подходящий объект. Оба примера показаны в листинге 6-13.

Листинг 6-13. Поддержка различных типов аргументов

```
/**
* @param string|array $emailAddresses
*/
public function setTo($emailAddresses): void
{
if (!is_array($emailAddresses)) {
$emailAddresses = [$emailAddresses];
}
// ...
}
/**
* @param int|DateTimeImmutable $time
*/
public function setLastModified($time): void
{
if (is_int($time)) {
$time = DateTimeImmutable::createFromFormat('U', $time);
}
if (!$time instanceof DateTimeImmutable) {
throw new \InvalidArgumentException(...);
}
// ...
}
```

Используйте фабрики объектов

Вероятно, между различными версиями пакета класс будет иметь разные зависимости. Рассмотрим класс `Validator` из листинга 6-14. Начальная версия этого класса вообще не требует аргументов конструктора. Более поздняя версия стала «интернациональной» и, следовательно, требует внедрения службы `Translator`.

Листинг 6-14. В следующей версии класс Validator имеет дополнительный аргумент конструктора

```
// Версия 1.0.0, аргументов конструктора нет;
class Validator
{
public function __construct()
{
// ...
}
}
// Применение:
$validator = new Validator();
// Версия 2.0.0, добавлен один аргумент конструктора;
class Validator
{
public function __construct(Translator $translator)
{
// ...
}
}
// Применение:
$validator = new Validator(new Translator());
```

Когда пользователи обновляют пакет валидатора с версии 1.0.0 до 2.0.0, работа их приложения будет прервана, потому что они еще не предоставили этот дополнительный аргумент конструктора.

Чтобы предотвратить подобные нарушения обратной совместимости, было бы лучше, если бы мы изначально предоставили фабрику для объектов `Validator`. Таким образом, пользователям нужно только создать фабрику (которая не требует аргументов конструктора), после чего они могут использовать ее для создания нового валидатора (*см.* листинг 6-15).

Листинг 6-15. Класс ValidatorFactory

```
class ValidatorFactory
{
public function createValidator(): Validator
```

```
{
$translator = new Translator();
return new Validator($translator);
}
}
// Применение:
$validator = (new ValidatorFactory())->createValidator();
```

Если в дальнейшем классе `Validator` понадобится какой-либо другой аргумент конструктора, фабрика негласно добавит его, и пользователю не нужно будет об этом знать. Опять же, эта хитрость называется «инкапсуляция»: на этот раз класс `ValidatorFactory` инкапсулирует логику создания Validator.

И так далее…

К настоящему времени вы, вероятно, уловили суть. Есть много способов сделать ваш код обратно совместимым и по-прежнему учитывать будущие изменения. Если рассматривать эту тему более подробно, я хотел бы обратить ваше внимание на статью Гарретта Руни «Сохранение обратной совместимости»[1]. В ней он описывает множество интересных способов, с помощью которых разработчики проекта Subversion пытались поддерживать обратную совместимость, одновременно обеспечивая *прямую совместимость*. Еще один интересный документ от команды разработчиков фреймворка Symfony, который также может стать хорошим руководством для вас, можно найти на странице https://symfony.com/doc/current/contributing/code/bc.html. Одно из последних предложений: для PHP есть утилита Roave Backward Compatibility Check (https://github.com/Roave/BackwardCompatibilityCheck), которая может анализировать код в репозитории и выяснить, существуют ли в нем какие-либо нарушения обратной совместимости.

Она предоставляет подробный обзор изменений API вашего пакета и того, как эти изменения могут нарушить клиентский код. Это может сэкономить вам много времени и избавить от разочарований в дальнейшем.

Наконец, существует метаперспектива, которую вы должны учитывать при выполнении всей этой работы, чтобы предотвратить

---

[1] *Garrett Rooney*. Preserving Backward Compatibility // https://web.archive.org/web/20180121015221/http://www.onlamp.com/pub/a/onlamp/2005/02/17/backwardscompatibility.html (первоначальная страница больше не доступна).

нарушения обратной совместимости. Если при работе с вашими пакетами вам важна *обратная совместимость*, вам будет очень трудно двигаться вперед. Как говорит Энтони Феррара:

> *[…] с каждым релизом у вас появляется все больше хлама. Со временем это создает эффект остановки для задействованной кодовой базы, что делает почти невозможным очистку и «улучшение». […] Поэтому в следующий раз, когда вы захотите предложить изменение, а не думать о том, как оно может нарушить обратную совместимость, попробуйте подумать, как сделать это изменение совместимым с будущими вариантами использования и изменениями. Лучший способ предотвратить нарушения обратной совместимости – это предвидеть их с самого начала*[1].

Добавьте метафайлы

Метафайлы, которые абсолютно необходимы, представляют собой краткое руководство в виде файла `README` и некоторые юридические материалы в виде файла лицензии.

Файл README и документация

Файл `README` должен находиться в корневом каталоге пакета. В нем содержится все, что нужно пользователю для начала работы. Файл `README` может быть единственной официальной документацией для пакета. В противном случае он должен содержать ссылку на другой источник документации внутри пакета (например, в его каталоге `docs`) или отдельный веб-сайт. Какую бы стратегию вы ни выбрали, файл `README` обязателен, поскольку он является отправной точкой для тех, кто хочет узнать больше о вашем пакете.

Файл `README` – это текстовый файл. Строки должны быть обернуты на соответствующую ширину (например, 80 столбцов), чтобы его можно было читать в терминале. Также было бы неплохо применить к нему стилизацию и структурирование. Довольно традиционным считается писать этот файл, используя язык Markdown[2],

---

1 *Anthony Ferrara*. Backwards Compatibility Is For Suckers // https://blog.ircmaxell.com/2013/06/backwards-compatibility-is-for-suckers.html.

2 https://daringfireball.net/projects/markdown/syntax.

который дает некоторые основные параметры разметки. Вы можете написать несколько слов курсивом или полужирным шрифтом, добавить блоки кода и использовать заголовки разделов. Если вы используете в своем файле `README` Markdown, переименуйте файл в `README.md`.

Файл `README` должен содержать как минимум следующие разделы.

Установка и настройка

Это может быть так же просто, как упомянуть команду, с помощью которой можно установить пакет в проект на языке PHP, например:

```
composer require matthiasnoback/some-package
```

Затем расскажите пользователям обо всем, что им нужно сделать, чтобы использовать пакет.

Возможно, им нужно настроить какие-то вещи, очистить кеш, добавить таблицы в базу данных и т. д.

Применение

Нужно показать пользователям, как они могут использовать код в вашем пакете. Это требует быстрого объяснения некоторых вариантов использования и *примеров кода* для этих ситуаций. Пакеты часто содержат отдельный каталог с (рабочим) образцом кода.

Точки расширения (не обязательно)

Если пакет предназначен для расширения, если для него есть плагины или комплекты/модули, облегчающие интеграцию библиотеки в проект на основе фреймворка, обязательно укажите эти *точки расширения*.

Ограничения (не обязательно)

Следует упомянуть случаи использования, для которых пакет в настоящее время не предлагает никакого решения. Вы также должны упомянуть известные проблемы (ошибки или другие ограничения) и, возможно, некоторые функции, которые вы собираетесь реализовать через какое-то время.

Лицензия

Еще один обязательный файл – это файл `LICENSE`. Даже если вы уже, возможно, указали *название* лицензии, применимой к вашему па-

кету, в файле определений пакета вы все равно должны добавить *полную* лицензию. Она должна находиться в файле с именем LICENSE в корне пакета. Очень популярная лицензия на программное обеспечение с открытым исходным кодом – лицензия MIT[1] (*см.* листинг 6-16).

Листинг 6-16. Лицензия MIT

```
Copyright (c) <year(s)> <name(s)>

Permission is hereby granted, free of charge, to any person
obtaining a copy of this software and associated documentation
files (the "Software"), to deal in the Software without
restriction, including without limitation the rights to use,
copy, modify, merge, publish, distribute, sublicense, and/or
sell copies of the Software, and to permit persons to whom
the Software is furnished to do so, subject to the following
conditions:

The above copyright notice and this permission notice shall
be included in all copies or substantial portions of the
Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY
KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE
WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR
PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS
OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR
OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR
OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE
SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.
```

Если вас интересует, почему важно иметь в своем пакете файл лицензии, то это зависит от страны, в которой вы живете, но некоторым компаниям требуется ваше явное разрешение на использование вашего кода так, как они намереваются это сделать. Они хотят предотвратить юридически неудобные ситуации, вызванные случайным нарушением авторских прав. Не менее важно то, что этот файл избавляет вас от какого бы то ни было ущерба, который может нанести ваш код, если его будет использовать кто-либо еще.

Журнал изменений (не обязательно)

Помимо обязательных файлов README и LICENSE, вы также должны рассмотреть возможность добавления журнала изменений. Это

[1] https://opensource.org/licenses/MIT.

позволит пользователям легко узнать, что изменилось в процессе выхода других версий пакета. На основании данной информации они могут решить, нужно ли им или полезно обновлять установленную версию пакета.

Каждая новая версия (старшая, младшая или версия патча) получает свой собственный раздел в журнале изменений, в котором вы описываете изменения, сделанные с момента выхода предыдущей версии. Вы должны кратко описать новые функции, которые были добавлены, то, что устарело (но не было удалено), и проблемы, которые были исправлены, а также, возможно, указывают на ошибки в системе отслеживания ошибок. *См.* листинг 6-17, где приводится пример файла `CHANGELOG.md`.

Листинг 6-17. Выдержка из файла CHANGELOG.md

```
# Changelog
## v0.5.0
- Automatically resolve a definition's class before comparing
it to the expected class.
## v0.4.0
- Added `ContainerBuilderHasSyntheticServiceConstraint` and
corresponding assertion (as suggested by @WouterJ).
...
```

Стандартного формата этого файла не существует, хотя тот, что предлагается на сайте keepachangelog.com, является простым и полным.

Примечания касательно обновлений (не обязательно)

Каждый раздел в журнале изменений может содержать примечания относительно обновлений, которые сообщают пользователям, что им нужно делать при обновлении до более новой версии. Например, если какие-то классы устарели, рекомендуется указать в журнале изменений, в какой версии вы их удалили.

В некоторых случаях эти примечания начинают занимать слишком много места, что приводит к беспорядку в журнале изменений. Тогда приходит время переместить эти примечания в определенные файлы `UPGRADE-x`. Например, файл `UPGRADE-3.md` будет содержать инструкции по обновлению зависимости с версии 2.x.x до 3.x.x. Помните, что между старшими версиями никаких действий от пользователя не требуется, поскольку младшие версии и версии патча используют только обратно совместимые изменения.

Руководство по содействию (не обязательно)

В частности, если ваш проект является проектом с открытым исходным кодом, вы можете рассмотреть возможность добавления отдельного файла, в котором вы описываете процесс внесения вклада в пакет. Этот файл должен содержать такие вещи, как:

- предложения по установке пакета таким образом, чтобы вы могли запустить набор тестов и начать работу над новой функцией, исправлением ошибок и т. д.;
- требования к описанию запросов на принятие изменений, стилю кодирования и т. д.;
- как обратиться за помощью, где обсудить проблемы и т. д.

Контроль качества

Мы уже обсудили большое количество характеристик пакета, которые могли бы квалифицировать его как хороший пакет (или «хороший продукт»). Большинство этих характеристик были связаны с инфраструктурой пакета: пакет должен хорошо вести себя, когда дело касается системы управления версиями, файла определений пакета, зависимостей и их версий, обратной совместимости. Для использования пакета необходимо наличие нескольких метафайлов, таких как документация, файл лицензии и т. д.

Возможно, вы заметили, что до сих пор мы не уделяли много внимания фактическому коду в вашем пакете. Конечно, мы подробно обсудим необходимые характеристики классов в пакете в последующих главах. Но в последних разделах этой главы я сначала укажу на некоторые аспекты инфраструктуры пакетов, которые помогут вам создавать пакеты с кодом высокого качества.

Качество с точки зрения пользователя

Пакет дает некоторые неявные обещания относительно кода, который он содержит. По существу, он говорит: «Вы можете добавить меня в свой проект. *Мой* код удовлетворит *ваши потребности*. Вам не придется писать этот код самостоятельно. И это сделает вас очень счастливыми».

Когда я натыкаюсь на пакет, который может обеспечить необходимую мне функциональность, первым делом я читаю файл `README` (и, возможно, любую другую доступную документацию). Когда описание пакета напоминает мне мои собственные идеи касательно

кода, который я бы написал, если бы этого пакета не существовало, следующее, что я делаю, – это погружаюсь в код. Я быстро сканирую структуру каталогов, имена классов, а затем код внутри этих классов, которые затем критически оцениваю.

Прежде всего я ищу варианты использования, которые поддерживает пакет. Лицо, отвечающее за сопровождение пакета, вероятно, создало этот пакет для поддержки одного из своих конкретных вариантов использования. Скорее всего, мой собственный вариант (немного, если не сильно) отличается. Поэтому одна из конкретных характеристик, которую я ищу, – это расширяемость: можно ли изменить поведение некоторых классов в пакете без фактического изменения самого кода? Хорошими признаками расширяемости являются использование интерфейсов и внедрение зависимостей.

Кроме того, код пакета, вероятно, содержит ошибки, которые необходимо исправить. Пробираясь сквозь код, я пытаюсь оценить объем работы, необходимый для устранения любой проблемы с кодом – содержит ли пакет классы с очень большим количеством ответственностей? Можно ли заменить ошибочные реализации простой реализацией интерфейса, определенного в пакете, или мне придется долго копировать страницы кода, чтобы повторить его поведение (после чего я все равно решу избавиться от этого пакета)?

Наконец, я взгляну на автоматизированные тесты, которые есть в этом пакете. Достаточно ли их? Сами они состоят из чистого кода? Имеют ли они смысл или они здесь только для тестового покрытия? Что делать, если тестов нет вообще (что случается со *многими* пакетами)? Как я могу доверять этому коду и использовать его для работы в собственном проекте?

Как я вообще мог набраться смелости поместить этот код в производственный сервер и позволить реальным пользователям его выполнить?

Причина моей осторожности при добавлении зависимости в свой проект заключается в том, что как только он будет установлен и я начну использовать содержащийся в нем код, я буду нести за него ответственность[1]. Хотя многие очень серьезно относятся к оказанию поддержки для своих пакетов, не каждый всегда будет исправлять все проблемы, о которых сообщают, или добавлять все отсутствующие функции, даже если вы достаточно хороши, чтобы создать для нее запрос на принятие изменений.

---

[1] Игорь Видлер подробно обсуждает это в своей статье: https://igor.io/2013/09/24/dependency-responsibility.html.

Скорее всего, вы окажетесь в одиночестве, если пакет не соответствует вашим ожиданиям.

Таким образом, вы должны иметь возможность исправлять ошибки и добавлять функции в пакет, не изменяя код внутри пакета (поскольку на самом деле вы не можете это сделать). Вам должно быть комфортно работать с ним.

Что нужно сделать человеку, отвечающему за поддержку пакета

Будучи лицом, которое отвечает за поддержку пакета, вы должны писать код, следуя установленным принципам проектирования, таким как принципы SOLID, описанные в предыдущей части этой книги. Но есть и другие (гораздо более простые) рекомендации, которым нужно следовать, чтобы создать хороший, или «чистый», код.

Статический анализ

Чтобы убедиться, что качество кода имеет определенный уровень и не ухудшается со временем, вы можете использовать инструменты автоматического статического анализа. Эти инструменты могут проверять код и выносить вердикт, основанный на наборе правил, которые в большинстве случаев могут быть полностью настроены в соответствии с вашими собственными стандартами качества. Популярными средствами статического анализа для PHP являются PHPStan и такие интегрированные среды разработки, как PhpStorm от компании JetBrains. Однако существует множество других доступных инструментов, большинство из которых перечислено на сайте PHP Quality Assurance (https://phpqa.io).

Добавьте тесты

Конечно, важно, чтобы ваш код выглядел хорошо. Но еще важнее, чтобы он хорошо работал. А как это можно проверить? Добавив набор тестов «соответствующего размера» в свой пакет.

Существуют совершенно разные мнения по поводу того, что именно это означает: сколько тестов вы должны написать? Вы сначала пишете тесты[1], а потом код? Стоит ли добавлять интеграци-

[1] Роберт К. Мартин обсуждает, что означает подход «сначала тест», в своей статье: https://8thlight.com/blog/uncle-bob/2013/09/23/Test-first.html.

онные или функциональные тесты? Какой объем покрытия кода необходим вашему пакету?

Ключевой вопрос, который вы должны задать себе, заключается в следующем: *меня волнует будущее моего кода*? Тесты предназначены для последующего безопасного рефакторинга. Если вы просто пишете код и используете его в одном проекте, то вам может не потребоваться писать тесты для этого кода. Напротив, вам определенно понадобится довольно большой набор тестов, если этот код будет использоваться *кем-то еще* в *любом другом проекте*. В этом случае вы хотите продолжать исправлять ошибки или добавлять новые функции в пакет. Если у вас нет тестов, внесение этих изменений становится трудным и опасным делом. Как можно доверять пакету, чтобы он работал как положено после того, как вы внесли изменения?

Итак, тесты поддерживают рефакторинг. Они очень помогут вам предотвратить регрессию в будущих фиксациях. Но тесты также служат спецификацией вашего кода. Они описывают ожидаемое поведение, когда пользователь будет что-то делать с кодом в какой-то конкретной ситуации. Вот почему тесты теоретически могут служить документацией к коду.

Конечно, это не совсем так, потому что тесты рассказывают только кусочки истории, но не всю историю. У них нет введения, нет эпилога, и они не заполняют никаких (концептуальных) пробелов в знаниях. Тем не менее тесты как спецификация кода и описание его поведения важны, потому что они позволяют пользователям кода узнать, какие вызовы методов они могут сделать, какие аргументы они должны предоставить и какие предварительные условия необходимы, прежде чем смогут это сделать.

Если у пакета нет тестов или их слишком мало, он сообщает пользователям:

«Меня не волнует будущее этого кода. Я не уверен, что, когда я что-то изменю, все будет работать. На самом деле нет никакой надежды, что этот код надежен. Я использую его сегодня и не забочусь о завтрашнем дне».

Настройте непрерывную интеграцию

Все тесты должны выполняться часто. Конечно, вы постоянно запускаете тесты во время разработки. Но каждый раз, когда вы создаете другую ветку (для новой версии), исправляете ветки с помощью исправлений ошибок, или когда принимаете запросы на принятие изменений, вам приходится снова запускать тесты. Иначе

вы не будете знать наверняка, что все работает, как ожидалось. Делать все это вручную было бы слишком трудоемко. И здесь пригодится непрерывная интеграция.

Непрерывная интеграция означает, что каждое изменение в репозитории проекта будет запускать процесс его сборки. Если что-то пойдет не так, команда, работающая над проектом, получит уведомление и сможет (немедленно) устранить проблему.

Для программных продуктов, которые будут поставляться, процесс сборки может включать в себя создание исполняемого файла или файла в формате ZIP. Для большинства проектов процесс сборки в основном интересен, потому что будут запущены все тесты. Некоторые другие артефакты, которые могут быть получены в процессе сборки, – это покрытие кода и показатели качества кода.

Заключение

Большинство вещей, которые мы обсуждали в этой главе, носили очень практический характер. Основная причина этого состояла в том, что вам нужно правильно настроить инфраструктуру вашего пакета, прежде чем его можно будет использовать многократно. Чтобы вы, ваши товарищи по команде или другие разработчики со всего мира имели возможность использовать пакет в своих проектах, это должен быть *действительно хороший продукт*. Вы (или человек, отвечающий за поддержку пакета, который является вашим преемником) должны иметь возможность выпустить пакет один раз и поддерживать будущие выпуски посредством хорошей инфраструктуры. Пользователи должны понимать, что это за пакет, как можно его использовать и чего они могут ожидать от вас относительно будущих версий.

Код, выпускаемый как один пакет, представляет собой первый аспект *связности* пакета. Если процесс *выпуска* пакета неуправляем или вообще не управляется, его нельзя *использовать повторно* должным образом. Эта глава дала вам обзор того, что это означает для пакета, – быть выпущенным управляемым способом, от первого выпуска до любого последующего выпуска, от версий патча до старших версий. В этом процессе есть много деталей, которые мы не рассмотрели, но часто они специфичны для используемого вами языка программирования. Последнее замечание, прежде чем мы продолжим обсуждение второго принципа связности, принципа *совместного повторного использования*. Возможно, я напугал

вас, написав обо всем, что вам нужно сделать, чтобы создать «хорошие» пакеты. Может быть, у вас появилось искушение отложить эту книгу и отпустить свою мечту однажды опубликовать пакет, который будут использовать много, много людей. Но не сдавайтесь! Конечно, создание вашего первого пакета может доставить вам немало хлопот. Это займет некоторое время, вы можете чувствовать себя немного неуверенно в отношении шагов, которые вы делаете, вы можете забыть что-то, можете сделать ошибки. Хорошо то, что вы быстро научитесь, выработаете некую привычку, а, по моему опыту, другие люди не стесняются давать полезные отзывы, которые должны помочь вам со временем выпускать пакеты еще лучшего качества.

После того как вы закончите читать эту книгу, продолжайте искать практические предложения, чтобы делать вещи, которые были описаны более абстрактно в этой главе, и стать частью живого сообщества совместного использования кода для разработчиков.

Глава 7

Принцип совместного повторного использования

В главе 6 мы обсудили принцип *эквивалентности повторного использования и выпуска*. Это первый принцип связности пакетов: он дает важную информацию относительно того, какие классы в пакете связаны друг с другом, а именно классы, которые вы можете правильно выпустить и поддерживать в качестве пакета. Вы должны позаботиться о доставке пакета, который является реальным продуктом.

Если вы будете следовать всем советам, данным в предыдущей главе, у вас будет хороший пакет. Он будет очень удобен в использовании и легкодоступен, поэтому его быстро освоят другие разработчики. Но даже если пакет ведет себя хорошо, *как пакет*, в то же время он может быть не очень *полезным*.

Допустим, у вас есть прекрасная коллекция очень полезных классов, реализующих несколько интересных функций. Когда вы группируете эти классы в пакеты, следует избегать двух крайностей. Если вы выпускаете все классы как один пакет, то заставляете своих пользователей загружать весь пакет в свой проект, даже если они используют очень небольшую его часть, что довольно обременительно для них в плане поддержки.

С другой стороны, если вы поместите каждый отдельный класс в отдельный пакет, вам придется выпустить множество пакетов, что, в свою очередь, ляжет бременем на вас. В то же время пользователям сложно управлять своим списком зависимостей и отслеживать все новые версии этих крошечных пакетов.

В этой главе мы обсуждаем второй принцип связности пакетов, который называется принципом *совместного повторного использования*. Он помогает решить, какие классы следует объединить в пакет и, что более важно, какие классы нужно перенести в другой пакет.

Когда мы выбираем классы или интерфейсы для повторного использования, принцип *совместного повторного использования* говорит нам[1]:

Классы, которые используются вместе, упаковываются вместе.

Поэтому когда вы разрабатываете пакет, то должны поместить в него классы, которые будут использоваться *вместе*. Это может показаться немного очевидным; вы бы не поместили абсолютно несвязанные классы, которые *никогда* не используются вместе, в один пакет. Но все становится несколько менее очевидным, когда мы рассматриваем другую сторону этого принципа, – вы не должны помещать классы в пакет, которые *не* используются вместе. Сюда входят классы, которые, *вероятно, не* будут использоваться вместе (в результате чего у пользователя остается нерелевантный код, импортированный в его проект).

В этой главе мы рассмотрим пакеты, которые явно нарушают это правило: они содержат все виды классов, которые не используются вместе, – либо потому, что эти классы реализуют изолированные функции, либо потому, что у них разные зависимости. Иногда тот, кто отвечает за поддержку пакета, помещает эти классы в один пакет, потому что они имеют некое концептуальное сходство. Некоторые могут подумать, что это повышает удобство использования пакета для них или его пользователей.

В конце этой главы мы попытаемся сформулировать данный принцип в более позитивной форме и обсудим некоторые наводящие вопросы, которые можно применять, чтобы привести ваши пакеты в соответствие с принципом *совместного повторного использования*.

Существует много признаков, или «запахов», по которым можно распознать пакет, нарушающий принцип *совместного повторного использования*. Я расскажу о некоторых из этих признаков, используя в качестве примеров реальные пакеты. Короткое отступление, прежде чем мы продолжим, – я никоим образом не хочу оспаривать значимость этих пакетов. Это хорошо зарекомендовавшие себя пакеты, созданные опытными разработчиками и используемые многими людьми во всем мире. Однако я не согласен с некоторыми из вариантов разработки этих пакетов. Поэтому не

---

[1] *Robert C. Martin*. The principles of OOD // http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod.

следует принимать мои комментарии как гневную критику. Это жест в сторону того, что я считаю идеальным подходом к разработке пакетов.

Пласты функций

Наиболее важной характеристикой пакетов, нарушающих принцип общего повторного использования, является то, что я называю «пласты функций». Мне очень нравится термин *пласты*, и сейчас самое время его использовать. *Пласт* – это:

> *слой материала, естественно или искусственно сформированный, часто один из нескольких параллельных слоев, наложенных друг на друга.*

Я хотел бы определить «пласты функций» как функции, существующие вместе в одном пакете, но которые не зависят друг от друга. Это означает, что вы сможете использовать функцию A без функции B, но добавление функции B возможно без нарушения функции A. Это также означает, что впоследствии отключение функции B не станет проблемой и не приведет к выходу из строя функции A. Функции A и B не касаются друг друга; они работают *параллельно*.

В контексте пакетов пласты функций часто проявляют себя как классы, которые связаны друг с другом, потому что они реализуют некую специфическую функцию. Но после того, как одна из функций была реализована, ответственный за поддержку пакета продолжал добавлять новые функции, состоящие из конгломератов классов, в один и тот же пакет. В большинстве случаев это происходит потому, что функции связаны *концептуально*, а не *материально*.

Очевидное расслоение

Иногда многослойный пакет можно распознать по тому факту, что он буквально содержит разное пространство имен для каждого *пласта функций*. Подобных примеров много, но давайте посмотрим на пакет `symfone/security`[1], который содержит компонент Symfony Security[2]. Как вы можете видеть, посмотрев на его дерево

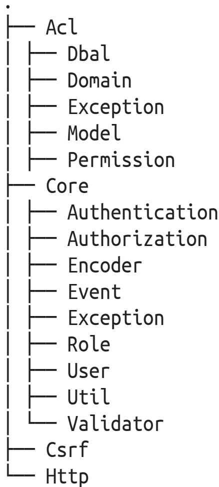
---

[1] https://packagist.org/packages/symfony/security.

[2] https://github.com/symfony/Security.

каталогов / пространств имен, у него есть четыре основных пространства имен – `Acl`, `Core`, `Csrf` и `http`, – каждое из которых содержит множество классов (*см.* листинг 7-1).

Листинг 7-1. Дерево каталогов пакета symfony/security



```
.
├── Acl
│  ├── Dbal
│  ├── Domain
│  ├── Exception
│  ├── Model
│  ├── Permission
├── Core
│  ├── Authentication
│  ├── Authorization
│  ├── Encoder
│  ├── Event
│  ├── Exception
│  ├── Role
│  ├── User
│  ├── Util
│  └── Validator
├── Csrf
└── Http
```

Классы из этих основных пространств имен не должны использоваться одновременно: классы в `Acl` зависят только от `Core`, классы в `Http` зависят от классов в `Core` и, необязательно, от классов в `Csrf`. А не наоборот. Это означает, что если кто-то установит этот пакет для использования классов `Csrf`, он будет использовать только небольшое подмножество из этого пакета. Поэтому если вспомнить о том, что такое принцип *совместного повторного использования* («Классы, которые используются вместе, упаковываются вместе»), налицо явное нарушение этого принципа.

Как видно, некоторые из основных пространств имен разделены на второстепенные пространства. И, как выясняется, многие классы из второстепенных пространств имен могут использоваться отдельно от классов в других пространствах имен. Это снова указывает на то, что принцип *совместного повторного использования* был нарушен и что пакет должен быть разделен. К счастью, те, кто отвечает за поддержку пакетов, уже решили разделить этот пакет на несколько других пакетов, так что теперь, по крайней мере, основные пространства имен имеют собственный файл определений и могут быть установлены отдельно.

Скрытое расслоение

Во многих других случаях пакет имеет пласты функций, которые не так легко обнаружить. Классы, которые сгруппированы вокруг определенной функциональности, разделены не пространством имен, а по некоему другому принципу, например по *типу* класса. Посмотрим на пакет `nelmio/security-bundle`[1], содержащий `NelmioSecurityBundle`[2], который можно использовать в проектах Symfony для добавления специфичных мер безопасности, которые не предусмотрены самим фреймворком. Из-за природы компонента `HttpKernel`[3] все, что связано с безопасностью, может быть реализовано в качестве слушателя событий, который подключается к процессу преобразования запроса в ответ. Некоторые слушатели событий, связанные с безопасностью, будут препятствовать дальнейшей обработке ядром текущего запроса (например, на базе используемого протокола), а некоторые слушатели изменяют окончательный ответ (например, шифруют куки или данные сеанса).

Глядя на дерево каталогов этого пакета, можно легко определить, что большинство функций представлены в качестве слушателей событий, некоторые из которых могут использовать служебные классы, такие как `Encrypter` и `Signer` (*см.* листинг 7-2).

Листинг 7-2. Структура каталогов пакета nelmio/security-bundle

```
.
├── ContentSecurityPolicy
│ └── ContentSecurityPolicyParser.php
├── Encrypter.php
├── EventListener
│ ├── ClickjackingListener.php
│ ├── ContentSecurityPolicyListener.php
│ ├── EncryptedCookieListener.php
│ ├── ExternalRedirectListener.php
│ ├── FlexibleSslListener.php
│ ├── ForcedSslListener.php
│ └── SignedCookieListener.php
├── Session
│ └── CookieSessionHandler.php
└── Signer.php
```

[1] https://packagist.org/packages/nelmio/security-bundle.
[2] https://github.com/nelmio/NelmioSecurityBundle.
[3] https://github.com/symfony/http-kernel.

Как вы могли догадаться по именам слушателей, каждый слушатель обладает определенной функциональностью, и каждый из них может использоваться *отдельно*. Это предположение можно подтвердить, посмотрев доступные параметры конфигурации для этого пакета (*см.* листинг 7-3).

Листинг 7-3. Доступные параметры конфигурации для пакета nelmio/security-bundle

```
nelmio_security:
# signs/verifies all cookies
signed_cookie:
 names: ['*']
# encrypt all cookies
encrypted_cookie:
 names: ['*']
# prevents framing of the entire site
clickjacking:
 paths:
'^/.*': DENY
# prevents redirections outside the website's domain
external_redirects:
 abort: true
 log: true
# prevents inline scripts, unsafe eval, external
# scripts/images/styles/frames, etc
csp:
 default: [ self ]
...
```

Становится ясно, что все эти слушатели представляют разные функциональные возможности, которые можно настроить самостоятельно, и любого из слушателей можно отключить, в то время как другой по-прежнему будет работать. Это заставляет нас сделать вывод, что когда вы используете один из классов из этого пакета, вы не всегда будете использовать все остальные (и даже большинство других) классы внутри этого пакета. Таким образом, этот пакет совершенно явно нарушает принцип *совместного повторного использования*. Тому, кто хочет использовать только одну из функций, предоставляемых этим пакетом (а это возможно!), все равно нужно установить весь комплект.

Если мы посмотрим на остальные параметры конфигурации, где кажется, что некоторые фрагменты этого пакета, даже взаимоисключающие, становится немного интереснее: обработка HTTPS-запросов не может быть «принудительной» и «гибкой» одновременно (*см.* листинг 7-4).

Листинг 7-4. Обработка HTTPS-запросов не может быть «принудительной» и «гибкой» одновременно

```
nelmio_security:
...
# forced HTTPS handling, don't combine with flexible mode
# and make sure you have SSL working on your site before
# enabling this
# forced_ssl:
# hsts_max_age: 2592000 # 30 days
# hsts_subdomains: true
# flexible HTTPS handling
# flexible_ssl:
# cookie_name: auth
# unsecured_logout: false
```

Это означает, что когда вы используете один конкретный класс в этом пакете, то есть `ForcedSslListener`, вы определенно не будете использовать *все* остальные классы этого пакета. На самом деле вы *наверняка* не будете использовать класс `FlexibleSslListener`. Конечно, это определенно требует разделения пакета, поэтому пользователям не придется беспокоиться об этой исключительности.

Классы, которые можно использовать, только когда установлен...

Следует отметить, что в предыдущих примерах речь шла о пакетах, которые предоставляют *отдельные* пласты функций, но каждая из этих функций имеет *одинаковые зависимости* (в этом случае другие компоненты Symfony или весь фреймворк). Следующие примеры будут касаться пластов функций в пакетах, которые сами имеют разные зависимости.

Давайте посмотрим на пакет `monolog/monolog`, который содержит регистратор Monolog. Первичным классом этого пакета является класс `Logger` (очевидно). Однако реальная обработка сообщений журнала осуществляется экземплярами `HandlerInterface`. Сочетая разные обработчики, стратегии активации, средства форматирования и процессоры, можно настроить каждую часть сообщений регистрации. Пакет пытается обеспечить поддержку всего, что вы можете записывать в сообщения журнала, будь то файл, сервер журналов, база данных и т. д. В результате чего получается список файлов, приведенный в листинге 7-5 (он довольно большой, но все же короче реального списка).

Листинг 7-5. Список файлов в пакете monolog/monolog (сокращенный вариант)

```
.
├── Formatter
│ ├── ChromePHPFormatter.php
│ ├── FormatterInterface.php
│ ├── GelfMessageFormatter.php
│ ├── JsonFormatter.php
│ ├── LogstashFormatter.php
│ └── WildfireFormatter.php
├── Handler
│ ├── AmqpHandler.php
│ ├── BufferHandler.php
│ ├── ChromePHPHandler.php
│ ├── CouchDBHandler.php
│ ├── CubeHandler.php
│ ├── DoctrineCouchDBHandler.php
│ ├── ErrorLogHandler.php
│ ├── FingersCrossedHandler.php
│ ├── FirePHPHandler.php
│ ├── GelfHandler.php
│ ├── HandlerInterface.php
│ ├── HipChatHandler.php
│ ├── MailHandler.php
│ ├── MongoDBHandler.php
│ ├── NativeMailerHandler.php
│ ├── NewRelicHandler.php
│ ├── NullHandler.php
│ ├── PushoverHandler.php
│ ├── RavenHandler.php
│ ├── RedisHandler.php
│ ├── RotatingFileHandler.php
│ ├── SocketHandler.php
│ ├── StreamHandler.php
│ ├── SwiftMailerHandler.php
│ ├── SyslogHandler.php
│ ├── TestHandler.php
│ └── ZendMonitorHandler.php
├── Logger.php
└── Processor
├── MemoryProcessor.php
├── ProcessIdProcessor.php
└── PsrLogMessageProcessor.php
```

Разработчик может установить этот пакет и приступить к созданию экземпляров классов обработчиков для любого средства хранения, которое *уже доступно в его среде разработки*. Как поль-

зователю вам не нужно думать о том, какой пакет нужно установить; это всегда основной пакет. Казалось бы, это имеет высокий коэффициент использования: разве не просто? В какой бы ситуации вы ни находились, просто установите пакет `monolog/monolog`, и вы сможете сразу же его использовать (это называется подходом «батарейки прилагаются»).

Как мы увидим, такой вариант разработки сильно усложняет жизнь пользователю и тому, кто отвечает за поддержку пакета. Дело в том, что этот пакет не совсем честен относительно своих зависимостей. Он содержит код для всего, но всему этому коду необходимо много разных дополнительных вещей для правильной работы. Например, `MongoDBHandler` нужно установить расширение `mongo`. Теперь, когда я устанавливаю `monolog/monolog`, диспетчер пакетов не будет проверять, установлено ли расширение, потому что оно указано как *необязательная* зависимость (`ext-mongo` под ключом `suggest`) в файле определений пакета (*см.* листинг 7-6).

Листинг 7-6. Необязательные зависимости для пакета monolog/monolog

```
{
"name": "monolog/monolog",
...
"require": {
"php": ">=5.3.0",
"psr/log": "~1.0"
},
...
"suggest": {
"mlehner/gelf-php": "Send log messages to GrayLog2",
"raven/raven": "Send log messages to Sentry",
"doctrine/couchdb": "Send log messages to CouchDB",
"ext-mongo": "Send log messages to MongoDB",
"aws/aws-sdk-php": "Send log messages to AWS services"
...
},
...
}
```

Почему отвечающий за поддержку `monolog/monolog` не добавил зависимость `ext-mongo` в список необходимых пакетов? Представьте себе разработчика, который хочет использовать только `StreamHandler`, который просто добавляет сообщения журнала в файл в локальной файловой системе. Ему не нужна ни база данных Mongo, ни расширение `mongo`. Если `ext-mongo` будет необходимой зависимостью, установка пакета `monolog/monolog` вынудит его также уста-

новить расширение `mongo`, даже если код, которому действительно нужно это расширение, так и не будет выполнен в его среде. Вот почему `ext-mongo` – это просто *предложенная зависимость*.

Поэтому если я хочу использовать `MongoDBHandler` для хранения сообщений журнала в базе данных Mongo, то должен вручную добавить `ext-mongo` в качестве зависимости в свой проект (как показано в листинге 7-7). Это заставит диспетчера пакетов проверить, действительно ли расширение `mongo` было установлено.

Листинг 7-7. Добавление ext-mongo вручную в качестве зависимости проекта

```
{
"require": {
"ext-mongo": ...
}
}
```

Первая проблема для меня заключается в том, что я не знаю, какую версию расширения `mongo` мне нужно установить, чтобы использовать `MongoDBHandler`. Нет никакого способа узнать это, кроме как просто попытаться установить последнюю стабильную версию и *надеяться*, что она поддерживается `MongoDBHandler`. Я буду знать это наверняка, только если `MongoDBHandler` будет поставляться с тестами, которые полностью определяют его поведение. Тогда я могу выполнить эти тесты и посмотреть, будут ли они пройдены с конкретной версией `ext-mongo`, которую я только что установил. Это мое первое возражение против варианта разработки, который делает `MongoDBHandler` частью базового пакета Monolog.

Второе возражение против этого подхода состоит в том, что он заставляет меня добавлять расширение `mongo` в список зависимостей *моего собственного проекта*. Это неправильно, поскольку это не зависимость *моего* проекта, а зависимость пакета `monolog/monolog`. Это зависимость класса *внутри этого пакета*. В моем собственном проекте может вообще не быть никакого кода, связанного с MongoDB, но мне необходимо расширение `mongo`, поскольку я хочу использовать класс `MongoDBHandler`.

Таким образом, пакет `monolog/monolog` плохо обрабатывает свои зависимости, и я должен сделать это сам. Но это не совсем соответствует идее диспетчера пакетов, которому я даю указание устанавливать каждую мою зависимость и любые зависимости, необходимые им. Я хочу, чтобы все устанавливаемые мной пакеты *полностью заботились о своих зависимостях*.

Я не обязан знать зависимости каждого класса внутри пакета и гадать, какие из них мне нужно установить вручную, чтобы иметь возможность их использовать. Кроме того, я не должен выяснять, какая версия зависимости совместима с кодом в пакете. Это задача ответственного за поддержку пакета.

Данное обоснование относится к каждому из конкретных обработчиков, которые поставляет пакет `monolog/monolog`. И почти все обработчики будут использоваться исключительно любым конкретным пользователем, а это означает, что пакет нарушает принцип *совместного повторного использования* равное количество раз. Что касается любого обработчика в пакете, другие обработчики и их необязательные зависимости, вероятно, никогда не будут использоваться одновременно.

Предлагаемый рефакторинг

Решение этой проблемы простое – разделить пакет `monolog/monolog`. У каждого обработчика должен быть свой пакет с собственными *подлинными* зависимостями. Например, `MongoDBHandler` будет находиться в пакете `monolog/mongo-db-handler`, который можно определить, как показано в листинге 7-8.

Листинг 7-8. Файл определений для пакета monolog/mongo-db-handler

```
{
"name": "monolog/mongo-db-handler",
"require": {
"php": ">=5.3.2",
"monolog/monolog": "~1.6"
"ext-mongo": ">=1.2.12,<1.6-dev"
}
}
```

Таким образом, каждый конкретный пакет-обработчик может быть явным в отношении своих зависимостей и поддерживаемых версий этих зависимостей, и больше нет никаких необязательных зависимостей (подумайте: как может зависимость на самом деле быть «необязательной»?).

Если я приму решение установить `monolog/mongo-db-handler`, то могу быть уверен, что каждая зависимость будет проверена и что после установки пакета каждая строка кода внутри него будет исполняемой в моей среде разработки. На рис. 7-1 показано, как выглядит иерархия зависимостей после этого изменения.

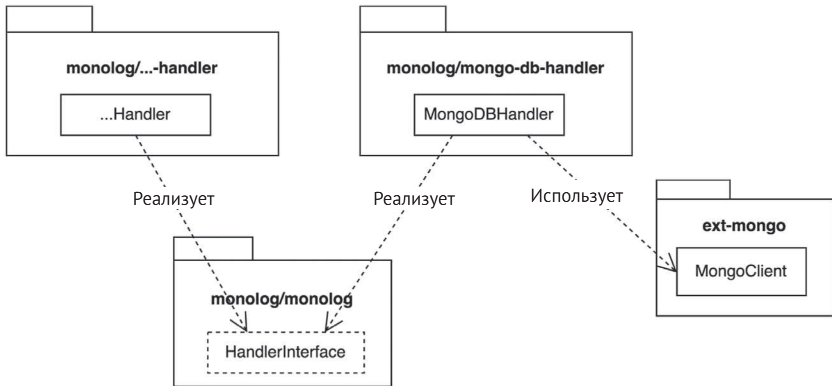


Рис. 7-1. Пакеты monolog с явными зависимостями

Ранее файл определений пакета `monolog/monolog` содержал полезные предложения относительно того, какие другие расширения и пакеты пользователь мог бы установить, чтобы разблокировать некоторые из его функций. Теперь, когда зависимость `ext-mongo` была перенесена в пакет `monolog/mongo-db-handler`, как пользователь узнает, что он может использовать базу данных Mongo, чтобы хранить сообщения журнала? Пакет `monolog/mongo-db-handler` можно указать в списке предлагаемых зависимостей пакета `monolog/monolog`, как показано в листинге 7-9.

Листинг 7-9. Предлагаемые зависимости для пакета monolog/monolog

```
{
"name": "monolog/monolog",
...
"suggest": {
"monolog/mongo-db-handler": "Send log messages to MongoDB",
"monolog/gelf-handler": "Send log messages to GrayLog2",
"monolog/raven-handler": "Send log messages to Sentry",
...
},
...
}
```

Пакет должен быть «компонуемым»

Давайте еще раз взглянем на `MongoDBHandler`, поскольку в настоящее время он по-прежнему является частью пакета `monolog/monolog`. Мы уже пришли к выводу, что после установки пакета вы не

сможете использовать этот класс без предварительной установки расширения `mongo`. Если мы этого не сделаем и попытаемся использовать этот конкретный обработчик, то получим все типы ошибок, в частности ошибки, связанные с классами, которые не были найдены. Код внутри `MongoDBHandler` синтаксически правильный, просто он не работает в контексте этого проекта.

Здесь необходимо делать концептуальное разделение, когда речь идет о правильности, разделение, которое нечасто делается разработчиками PHP. Во многих языках программирования проблемы с кодом могут возникать во время *компиляции* или во время *компоновки*. Компиляция кода означает проверку его синтаксиса, построение абстрактного синтаксического дерева и преобразование кода более высокого уровня в код более низкого уровня. Результатом компиляции являются объектные файлы, которые должны быть скомпонованы, чтобы быть исполняемыми. Во время процесса компоновки будут проверяться ссылки на классы и функции. Если функция или класс не определены ни в одном из объектных файлов, компоновщик выдает ошибку.

Одной из особенностей PHP является то, что в нем отсутствует процесс компоновки. Да, он компилирует код, но если класс или функция не существует, это будет замечено только во время выполнения, а в большинстве случаев даже в самый последний момент.

Я твердо верю, что хотя PHP позволяет нам быть очень гибкими в этом отношении, мы должны научить себя мыслить, как «компилируемый язык». Мы должны спросить себя: будет ли этот код компилироваться? Он определенно должен компилироваться, иначе это будет просто некорректный код. И относительно каждой явной, необязательной зависимости моего пакета: будет ли этот код «компоноваться»? Ответ будет «нет», если `MongoDBHandler` останется внутри пакета `monolog/monolog`, который имеет расширение `mongo` только в качестве предлагаемой зависимости. Если мы переместим `MongoDBHandler` в его собственный пакет `monolog/mongo-db-handler`, который имеет явную зависимость от `ext-mongo`, ответ будет «да», как и должно быть.

ИНСТРУМЕНТЫ СТАТИЧЕСКОГО АНАЛИЗА КАК ЗАМЕНА РЕАЛЬНОГО КОМПИЛЯТОРА

За последние несколько лет PHP как язык программирования продвигался все дальше и дальше в направлении к тому, чтобы стать

статистически типизованным языком. Код PHP останется динамическим языком с компиляцией во время выполнения. Но с каждым выпуском проверка типов улучшается. Кроме того, разработчики PHP компенсируют то, что язык не предлагает, используя все виды инструментов статического анализа. Эти инструменты становятся все более и более популярными, поскольку помогают выявить множество потенциальных проблем с кодом, прежде чем он будет запущен на сервере.

В контексте нашей дискуссии о компиляции и компоновке первого упоминания здесь заслуживает PHPStan:

> *PHPStan приближает PHP к компилируемым языкам в том смысле, что правильность каждой строки кода можно проверить перед запуском фактической строки*[1].

Сопоставимые инструменты – Psalm[2] и Phan[3].

Здесь стоит упомянуть еще один инструмент. Он называется Composer Require Checker[4] и проверяет, использует ли пакет импортированные символы (классы, интерфейсы и т. д.), которые не входят в его прямые зависимости. Это очень полезно, поскольку предотвращает сценарий, при котором ваш пакет использует класс, находящийся внутри пакета, который только *косвенно* является зависимостью вашего пакета. Другими словами, если пакет A зависит от пакета B, а B зависит от C, то если A также зависит от C, он должен сделать эту зависимость явной. В противном случае, если однажды B перестанет зависеть от C, A выйдет из строя. Если вы используете PhpStorm в качестве интегрированной среды разработки, плагин PHP Inspections также может сообщить вам о неявных зависимостях.

Чистые релизы

У пакета `monolog/monolog` есть еще одна особенность, которую я хотел бы обсудить здесь, что снова ведет нас в направлении создания отдельных пакетов для каждого конкретного обработчика.

1 https://github.com/phpstan/phpstan.
2 https://github.com/vimeo/psalm.
3 https://github.com/phan/phan.
4 https://github.com/maglnet/ComposerRequireChecker.

Как мы видели в предыдущей главе, важно, чтобы пакет был хорошим программным продуктом. Одной из важных характеристик хорошего программного обеспечения является то, что новые версии не вызывают нарушений обратной совместимости. Однако класс `MongoDBHandler` из пакета `monolog/monolog` демонстрирует явные признаки борьбы за обратную совместимость (*см.* листинг 7-10).

Листинг 7-10. Класс MongoDBHandler

```
class MongoDBHandler extends AbstractProcessingHandler
{
// ...
public function __construct(
$mongo,
$database,
$collection,
...
) {
if (!($mongo instanceof MongoClient
|| $mongo instanceof Mongo)) {
throw new InvalidArgumentException('...');
}
// ...
}
// ...
}
```

Первый параметр конструктора `$mongo` не имеет явного типа. Вместо этого валидность аргумента проверяется внутри конструктора, и эта проверка позволяет нам использовать два различных типа объектов `$mongo`. Это должен быть либо экземпляр `MongoClient`, либо экземпляр `Mongo`. Оба являются классами из расширения `mongo`, но класс `Mongo` устарел начиная с версии расширения 1.3.0.

Так что теперь внутри конструктора этого класса находится уродливый оператор `if`, который не позволяет строго типизировать аргумент `$mongo`, даже если в моем окружении установлена последняя версия расширения `mongo`. Это не должно быть необходимо. Мне бы хотелось, чтобы обработчик выглядел так, как показано в листинге 7-11.

Листинг 7-11. Класс MongoDBHandler со строго типизированным аргументом конструктора

```
class MongoDBHandler extends AbstractProcessingHandler
{
...
```

```
public function __construct(
MongoClient $mongo,
$database,
...
) {
// Дополнительной валидации не требуется;
// ...
}
// ...
}
```

Но если мы удалим оператор `if`, этот класс окажется бесполезным для тех, у кого в системе установлена более старая версия `mongo`.

Единственный способ решить эту дилемму – создать дополнительные ветви в репозитории пакета `monolog/monolog` в системе управления версиями – ветви для диапазонов версий MongoDB, которые должны получить специальное оформление, например `monolog/monolog@mongo_db_ older_than_1_3_0` и `monolog/monolog@mongo_db`. Конечно, скоро это закончится большим беспорядком. Пакет `monolog/monolog` имеет гораздо больше обработчиков, чем может потребоваться для такого оформления.

Давайте перенесемся в уже предложенное решение по переносу `MongoDBHandler` в его собственный пакет, файл определений которого выглядит так, как показано в листинге 7-12.

Листинг 7-12. Файл определений пакета monolog/mongo-db-handler (пересмотренный вариант)

```
{
"name": "monolog/mongo-db-handler",
"require": {
"php": ">=5.3.2",
"monolog/monolog": "~1.6"
"ext-mongo": ">=1.2.12,<1.6-dev"
}
}
```

Этот пакет `monolog/mongo-db-handler` размещен в отдельном репозитории, поэтому ему не нужно придерживаться версий основного пакета `monolog/monolog`. Это означает, что можно добавлять ветви, соответствующие различным версиям расширения `mongo`. Например, у вас может быть ветка 1.2.x и ветка 1.3.x, в соответствии с поддерживаемой версией расширения `mongo`. Тогда кто-то, у кого установлена версия 1.2 расширения `mongo`, может добавить версию 1.2 `monolog/monolog-dbhandler` в качестве зависимости в свой проект.

Листинг 7-13. Добавление версии 1.2 monolog/monolog-db-handler в виде зависимости

```
{
"require": {
"monolog/monolog-db-handler": "1.2.*"
}
}
```

Кто-то, у кого уже есть последняя версия расширения `mongo`, просто выберет `"~ 1.3"` в качестве ограничения версии.

Разделение пакета на основе его (необязательных) зависимостей выгодно не только пользователю пакета. Это также очень поможет человеку, отвечающему за поддержку. Он может позволить кому-то другому поддерживать пакет-обработчик, предназначенный для MongoDB, тому, кто уже пристально следит за выпусками расширений `mongo`. Этот человек не должен иметь возможность изменять основной пакет `monolog/monolog`. Этот пакет автоматически становится более стабильным, поскольку у него меньше причин для изменения (*см.* также главу 8, в которой рассказывается о принципе общей закрытости). Это само по себе хорошо для пользователей, которым не нужно отслеживать каждую новую версию, выходящую по причине изменения в одном из обработчиков, не используемых ими.

ЦЕНА РАЗДЕЛЕНИЯ

При обсуждении принципов разработки пакетов и их применении к реальным пакетам обычно рекомендуется разделять пакет на более мелкие. Хотя мы по-прежнему обсуждаем все причины, чтобы сделать это, приятно упомянуть, что тут есть компромисс. Чем меньше ваши пакеты, тем больше у вас их будет, тем больше работы вам придется потратить на создание новых выпусков, управление репозиториями, их проблемами, запросами на принятие изменений и т. д. Чем больше ваши пакеты, тем сложнее с ними работать с точки зрения пользователя. Пакет нужно будет часто обновлять, и пользователь будет загружать много кода и множество зависимостей, которые ему не нужны.

Хорошо иметь это в виду, когда вы разрабатываете пакеты. Вам нужно найти золотую середину между слишком большим количеством маленьких пакетов и слишком маленьким количеством больших пакетов.

Техническим решением, которое может быть полезным, является так называемое «монорепозиторий». Это означает, что код для

всех ваших пакетов будет размещен в одном репозитории. Любое изменение любого из пакетов будет зафиксировано в этом репозитории. Чтобы пользователи могли устанавливать каждый пакет отдельно, будет репозиторий только для чтения для каждого из подпакетов внутри монорепозитория.
Эти вложенные хранилища будут обновляться при каждом изменении основного репозитория. В Git данный процесс называется «subtree split». Нет необходимости настраивать его вручную, поскольку для этого существуют автоматизированные решения, такие как Git Subtree Split as a Service[1].

Бонусные функции

Мы рассмотрели очевидные и неочевидные пласты функций и то, почему они являются характеристиками пакетов, которые нарушают принцип *совместного повторного использования*. Иногда функции на самом деле не являются пластами, а представляют собой отдельные классы, которые тем не менее не принадлежат пакету. Давайте посмотрим на пакет `matthiasnoback/microsoft-translationator`[2], который создал я. Этот пакет содержит библиотеку MicrosoftTranslator[3], которая может использоваться для перевода текста с использованием API-интерфейса Microsoft (Bing) Translator. Переводчик зависит от пакета `kriswallsmith/buzz`[4], который содержит HTTP-клиента под названием `Buzz`[5]. Мой пакет использует свой класс `Browser` для выполнения HTTP-запросов к API-интерфейсам Microsoft OAuth и Translator, как вы можете догадаться по его (обнаженной) структуре каталогов.

Листинг 7-14. Сокращенная структура каталогов пакета matthiasnoback/microsoft-translation

```
.
├── Buzz
├── Exception
├── MicrosoftOAuth
└── MicrosoftTranslator
```

[1] https://www.subtreesplit.com.
[2] https://packagist.org/packages/matthiasnoback/microsoft-translator.
[3] https://github.com/matthiasnoback/microsoft-translator.
[4] https://packagist.org/packages/kriswallsmith/buzz.
[5] https://github.com/kriswallsmith/Buzz.

Разрабатывая эту библиотеку, я понял, что мое приложение может совершать многократные повторные вызовы API-интерфейса Microsoft Translator. Например, он несколько раз попросил службу перевода перевести слово «Submit» на нидерландский язык. И хотя при этом будет каждый раз вызываться новый HTTP-запрос, ответ от API всегда будет одним и тем же. Чтобы предотвратить повторяющиеся запросы, я решил добавить в пакет слой кеширования и подумал, что было бы неплохо сделать это, обернув клиент-браузер Buzz в класс `CachedClient`[1]. Класс `CachedClient` проанализирует каждый входящий запрос и заглянет в кеш, чтобы узнать, делал ли он этот запрос раньше. Если это так, будет возвращен кешированный ответ; в противном случае запрос будет перенаправлен реальному клиенту Buzz, а после этого новый ответ будет сохранен в кеше.

На рис. 7-2 приведена диаграмма зависимостей пакета `microsoft-translator`.

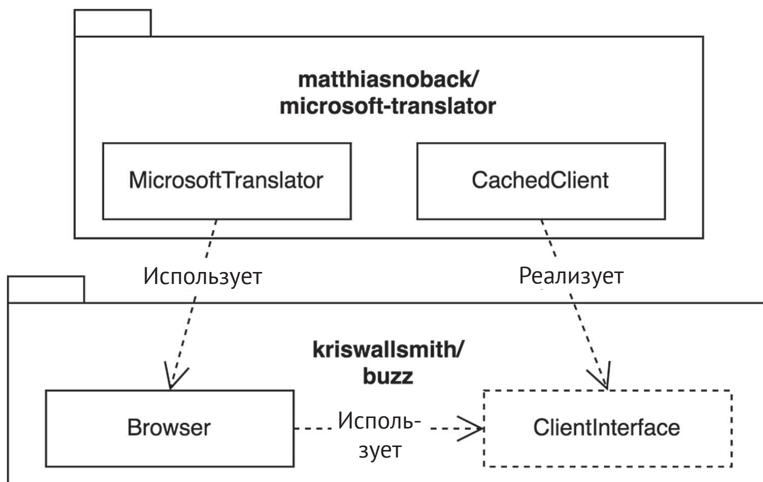


Рис. 7-2. Диаграмма зависимостей пакета microsoft-translation

Хотя в то время я думал, что дизайн у этой библиотеки довольно хороший, теперь я постараюсь устранить зависимость от Buzz. В Buzz нет ничего настолько особенного, что этому пакету *действительно* понадобится. На самом деле все, что ему нужно, –

[1] https://github.com/matthiasnoback/microsoft-translator/blob/v0.6.1/src/MatthiasNoback/Buzz/Client/CachedClient.php.

это «какой-нибудь HTTP-клиент». Учитывая необходимость абстракции, я бы использовал интерфейс для HTTP-клиентов (например, `HttpClientInterface`), а затем создал бы отдельный пакет под названием `matthiasnoback/microsoft-translationator-buzz`, который обеспечит реализацию моего собственного интерфейса `HttpClientInterface`, использующего Buzz. Более того, я мог бы полагаться на существующую абстракцию для HTTP-клиентов (например, HTTPlug[1]) или какой-либо другой стандартизированный и широко поддерживаемый интерфейс, такой как PHP Standard Recommendation (PSR)[2].

Но с дизайном этой библиотеки есть еще одна проблема: она содержит этот маленький умный класс `CachedClient`. Поскольку это действительно такой полезный класс, каждый пользователь этого пакета, вероятно, будет использовать его вместе с другими классами в пакете. Таким образом, здесь нет немедленного нарушения принципа *совместного повторного использования*. Однако предположим, что ваш проект уже зависит от пакета `kriswallsmith/buzz` и вам нужна реализация кеша для клиента-браузера Buzz. Пакет `matthiasnoback/microsoft-ranslationator` содержит такую реализацию, поэтому просто нужно установить его и использовать только его класс `CachedClient`, и никакой другой класс из того же пакета. Теперь становится кристально ясно, что этот пакет действительно нарушает принцип *совместного повторного использования*. Если вы используете класс из этого пакета, то не будете использовать *все* остальные классы.

Предлагаемый рефакторинг

Как вы уже догадались, чтобы решить эту проблему, нужно извлечь класс `CachedClient` и поместить его в другой пакет, `matthiasnoback/cached-buzz-client`, который не имеет ничего общего с Microsoft Translator и зависит только от `kriswallsmith/buzz`. Таким образом, любой может установить этот пакет в свой проект и использовать слой кеширования для клиентов Buzz. Более того, этот пакет может развиваться отдельно от пакета `microsoft-translation`. Другие люди могут способствовать этому, добавляя в него функции или исправляя ошибки. Эти улучшения станут доступны

---

1 https://github.com/php-http/httplug.
2 https://www.php-fig.org/psr/.

всем, кто зависит от пакета `cached-buzz-client`. При установке пакета `microsoft-translationator` пакет `cached-buzz-client` может быть предлагаемой, необязательной зависимостью (*см.* рис. 7-3).

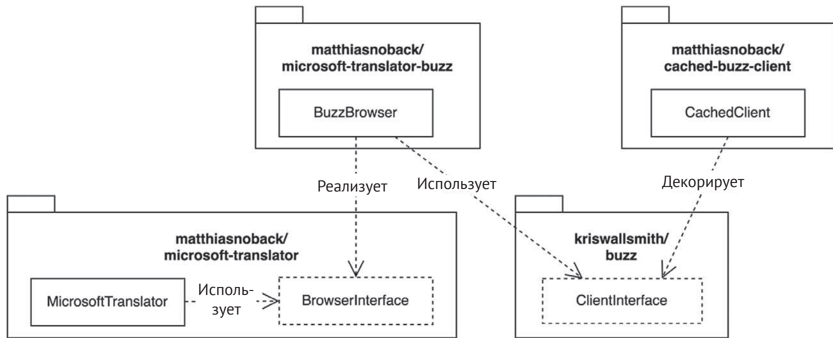


Рис. 7-3. Диаграмма зависимостей пакета microsoft-translationator после перемещения класса CachedClient

Просто здорово, что в Buzz теперь есть способ кеширования HTTP-ответов, и такой функционал действительно может представлять интерес для других людей. Однако если бы мы более тщательно подумали, прежде чем приступить к расширению самого HTTP-клиента Buzz, мы бы поняли, что гораздо более простое решение находится совсем рядом. Перефразируя функциональные требования, такие как «возможность кешировать результаты перевода», все, что нам нужно, – это расширить функциональность самого класса переводчика. Мы могли бы сделать это, используя ранее продемонстрированную технику декорирования класса с применением композиции. Для начала нам нужен интерфейс для класса переводчика. Затем мы создадим новый класс, который реализует этот интерфейс *и* получает экземпляр данного интерфейса в качестве аргумента конструктора. *См.* листинг 7-15, где приводится результат. Продемонстрированная методика кеширования возвращаемого значения функции называется «мемоизацией».

Листинг 7-15. Оборачиваем переводчик и кешируем возвращаемые значения метода translate()

```
interface TranslatorInterface
{
public function translate(string $text, string $to): string;
}
```

```
final class MicrosoftTranslator implements TranslatorInterface
{
public function translate(string $text, string $to): string
{
// Выполняем вызов к удаленному веб-сервису;
}
}
final class CachedTranslator implements TranslatorInterface
{
private $results = [];
public function __construct(TranslatorInterface $translator)
{
$this->translator = $translator;
}
public function translate(string $text, string $to): string
{
if (!isset($this->results[$text][$to])) {
$result = $this->translator
->translate($text, $to);
$this->results[$text][$to] = $result;
}
return $this->results[$text][$to];
}
}
$translator = new CachedTranslator(new MicrosoftTranslator());
// Обращаемся к удаленному сервису:
$translator->translate('Submit', 'nl');
// Следующий вызов будет использовать кешированный результат:
$translator->translate('Submit', 'nl');
```

Класс `CachedClient`, который мы только что обсуждали, является хорошим примером «бонусной функции», которая состоит только из одного класса. Кажется излишним устанавливать целый пакет со множеством классов в вашем проекте, просто используя один класс. Но, конечно же, это шкала ползунка. Какое количество или процентное соотношение классов будет приемлемым, чтобы пакет оставался неизменным и не был разбит на несколько пакетов?

Наводящие вопросы

Эти вопросы помогут вам решить для каждого класса, должен ли он быть в пакете, над которым вы работаете. На практике я обычно создаю класс внутри пакета, над которым уже работаю. После этого я могу решить переместить его в другой пакет, основываясь на следующих наводящих вопросах:

- *Использует ли этот класс зависимость?*

Если да, это необязательная/рекомендуемая зависимость? Тогда вы должны создать новый пакет, содержащий этот класс, явно указав его зависимость.

- *Будет ли этот класс полезен без остальной части пакета?*

Если да, то вам нужно создать новый пакет, чтобы пользователи могли получать только тот код, который они хотят применять.

Задавая себе два этих вопроса и следуя советам, которые они дают, вы автоматически разделите ваши пакеты в соответствии с их зависимостями и предоставляемыми ими функциями. Эти аспекты также являются основными причинами, по которым люди выбирают определенный пакет, а не другой – независимо от того, предлагает ли он слишком много или слишком мало необходимой им функциональности и совместимы ли его зависимости с зависимостями их собственных проектов. Если какой-либо из этих аспектов не соответствует требованиям пользователей, они не будут устанавливать пакет.

Когда применять этот принцип

Можно применять принцип *совместного повторного использования* в разные моменты жизненного цикла разработки пакета, например когда вы создаете новый пакет для существующих классов. Первое, что вам нужно сделать, – это сгруппировать классы, которые *всегда используются вместе*, и поместить их в пакет. Когда вам нужен класс A, а ему всегда нужен класс B, было бы плохой идеей помещать эти классы в разные пакеты `package-a` и `package-b`. Разделение этих классов потребует от разработчика, который хотел бы использовать класс A, установки обоих пакетов.

Но принцип *совместного повторного использования* также должен постоянно применяться при добавлении новых классов в существующий пакет. Вы должны проверить, будет ли новый класс, который вы собираетесь добавить, *всегда* использоваться вместе со всеми другими классами в пакете.

Скорее всего, добавляя в пакет новый класс, вы добавляете в пакет функции, которые используются не всеми, кому требуется пакет в качестве зависимости своего проекта.

Когда нарушать принцип

Как мы увидим в следующей главе, принцип *совместного повторного использования* – это принцип, который можно только максимизировать. Не *всегда* можно следовать ему идеально. Бывают моменты, когда вы можете поместить два класса в один пакет, которые *можно* использовать отдельно. Строго говоря, вы тем самым нарушаете принцип, но на то могут быть веские причины. Прежде всего ради удобства. Каждый пакет, который вы создаете, требует особого внимания. Он требует времени и энергии от вас как ответственного за его поддержку, и существует ограничение на количество пакетов, которые вы можете или хотите поддерживать.

Еще одна причина нарушить этот принцип заключается в том, что вы можете знать все о вашей целевой аудитории. Когда вы знаете, что почти все разработчики, использующие ваш пакет, используют его в проекте, созданном с применением фреймворка Laravel, вы можете менее строго соблюдать принцип *совместного повторного использования* и добавить несколько классов в свой пакет, что может иметь смысл, только когда кто-то использует комплексный фреймворк Laravel. Эти классы обычно находятся в отдельном пакете, потому что теоретически могут применяться по отдельности, но на практике они всегда будут использоваться вместе, что позволяет поместить их в один и тот же пакет.

Почему не следует нарушать этот принцип

Однако в большинстве случаев есть следующая веская причина *не* нарушать принцип *совместного повторного использования*: каждый класс, входящий в ваш пакет, *подвержен изменениям*. Возможно, один из его методов содержит ошибку, или что-то в его функционале необходимо изменить.

Вероятно, его интерфейс нужно изменить, и появится нарушение обратной совместимости. Будучи пользователем пакета, вы должны следить за всеми изменениями и решать, обновляться ли до новой версии и когда это делать. На это нужно время, потому что после обновления пакета вам нужно проверить все части вашего проекта, которые используют классы из этого пакета. Может быть, у вас есть автоматические тесты для этого, а может, и нет.

Однако вы не можете отказаться от обновлений. Вы должны позаботиться о том, чтобы ваш проект зависел от последних стабильных версий всех пакетов, чтобы предотвратить проблемы

(в будущем), например зависимость от устаревших или даже заброшенных пакетов. Если пакет, от которого вы зависите, большой, содержит много классов и связан с различного рода вещами, его обновление будет довольно сложной задачей. Существует много точек соприкосновения между кодом в вашем проекте и кодом внутри пакета, а это означает, что изменения будут иметь много побочных эффектов.

Напротив, когда пакет, от которого вы зависите, маленький, вам будет легче отслеживать изменения и менее болезненно обновлять пакет. Точек соприкосновения будет меньше, и поэтому вероятность того, что обновление испортит ваш проект, значительно меньше.

Таким образом, вы очень помогаете пользователям своего пакета, когда делаете его небольшим и помещаете в него только те классы, которые они будут использовать. Тогда они смогут безбоязненно обновлять свои зависимости.

Заключение

Мы узнали много нового о принципе *совместного повторного использования*, и к настоящему моменту должно быть ясно, что есть несколько веских причин для разделения пакетов. Эти причины имеют преимущества как для пользователей, так и для отвечающих за поддержку. Пакет, который придерживается принципа *совместного повторного использования*, имеет следующие характеристики:

- он связный: все содержащиеся в нем классы примерно одинаковы. Пользователям не нужно устанавливать большой пакет только для использования одного класса или небольшой группы классов;
- у него нет «необязательных» зависимостей: все его зависимости являются подлинными требованиями; они упоминаются явно и имеют правильные диапазоны версий. Пользователям не нужно вручную добавлять дополнительные зависимости в свой проект;
- они используют инверсию зависимостей, чтобы сделать зависимости абстрактными, а не конкретными;
- в результате они открыты для расширения и закрыты для модификации.

Добавление или изменение альтернативной реализации означает не открытие пакета, а создание дополнительного.

Глава 8

Принцип общей закрытости

В предыдущей главе мы обсуждали принцип *совместного повторного использования*. Это был второй принцип связности пакетов, который гласит, что мы должны помещать классы в пакет, которые будут использоваться вместе с другими классами в пакете. Если пользователь хочет использовать класс или группу классов по отдельности, для этого требуется разделить пакет.

Третий принцип связности пакетов называется принципом *общей закрытости*. Он тесно связан с принципом *совместного повторного использования*, потому что дает вам другой взгляд на *гранулярность*: вы получите еще один ответ на вопрос о том, какие классы связаны друг с другом в пакете, а какие нет. Принцип гласит[1]:

Классы в пакете должны быть закрыты относительно изменений одного и того же типа. Изменение, затрагивающее пакет, затрагивает все классы в этом пакете.

Таким образом, «общая закрытость» на самом деле означает *закрытость* относительно однотипных изменений.

Что касается кода в пакете, это означает, что когда что-то нужно изменить, вполне вероятно, что запрошенное изменение повлияет *только на один* пакет. Кроме того, когда требование изменяется и оно влияет на один пакет, оно, вероятно, повлияет на *все* классы в этом пакете.

Основное обоснование этого принципа заключается в том, что мы хотим, чтобы изменение было ограничено наименьшим количеством возможных пакетов. Люди, которые добавили ваш пакет в качестве зависимости к своему проекту, вероятно, будут следить за

---

[1] *Robert C. Martin*. Engineering Notebook. C++ Report, Nov-Dec, 1996 (PDF-версия доступна по адресу: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod).

новыми выпусками этого пакета, чтобы поддерживать весь код в своем проекте в актуальном состоянии. Когда появляется новая версия пакета, пользователь обновит свой проект, чтобы потребовать новую версию. Но он хочет делать это, только если сделанные *вами* изменения в пакете связаны с тем, как пакет используется в *его* проекте, поскольку каждое обновление требует от него проверки того, что его код по-прежнему корректно работает с новой версией вашего пакета.

Будучи ответственным за поддержку пакета, вы должны следовать принципу *общей закрытости*, чтобы не допустить «открытия» пакета по разным причинам. Это помогает предотвратить выпуск новых релизов, которые не имеют отношения к большинству ваших пользователей. С этой целью данный принцип рекомендует вам помещать классы в разные пакеты, если у них есть разные причины для изменения.

Эти причины можно разделить на несколько типов, каждый из которых мы обсудим в следующих разделах.

Изменение в одной из зависимостей

Рассмотрим пакет, в котором по-прежнему используются устаревшие PHP-функции `mysql_ *` для взаимодействия с базой данных MySQL. Вы решаете, что хотите удалить все вхождения этих функций в пакете и использовать вместо них намного лучшую альтернативу, PDO[1]. Вы приступаете к работе, но вскоре становится ясно, что из *всех классов* внутри пакета нужно изменить только два. Обычно это было бы хорошо, но в контексте разработки пакетов это означает, что пакет нарушает принцип *общей закрытости* – не все файлы внутри пакета закрыты относительно изменений одного и того же типа. Только некоторые классы затронуты недавно измененными требованиями, в то время как многие классы – нет. Другими словами, большинство классов закрыты относительно изменений, связанных с управлением базой данных, а некоторые – нет.

Классы, которые были изменены вместе, должны были находиться в отдельном пакете.

Этот пакет будет содержать все классы, которые будут затронуты одинаковыми изменениями. Он будет содержать классы, отвечающие за связь с базой данных MySQL. Другие пакеты будут содержать оставшиеся классы, которые не имеют ничего общего с конкретным способом доступа к базе данных.

---

[1] https://phptherightway.com/#databases.

Assetic

Реальным примером пакета, который содержит классы, закрытые относительно множества различных видов изменений, является пакет `kriswallsmith/assetic`[1], который содержит библиотеку Assetic[2], используемую для управления веб-ресурсами (например, объединение, сжатие и фильтрация файлов JS и CSS).

Глядя на структуру каталогов (*см.* листинг 8-1; я удалил много файлов, чтобы картина была яснее), видно, что она содержит основные классы, а также много классов, именуемых «фильтрами», которые используются для изменения содержимого файла ресурса (например, чтобы скомпилировать Less в CSS, сжать JS и т. д.).

Листинг 8-1. Дерево каталогов пакета kriswallsmith/assetic (сокращенная версия)

```
.
└── Assetic
├── Asset
├── Cache
├── Exception
├── Factory
├── Filter
│ ├── CoffeeScriptFilter.php
│ ├── CompassFilter.php
│ ├── CssEmbedFilter.php
│ ├── CssImportFilter.php
│ ├── CssMinFilter.php
│ ├── CssRewriteFilter.php
│ ├── DartFilter.php
│ ├── EmberPrecompileFilter.php
│ ├── FilterCollection.php
│ ├── FilterInterface.php
│ ├── GoogleClosure
│ ├── GssFilter.php
│ ├── HandlebarsFilter.php
│ ├── JpegoptimFilter.php
│ ├── JpegtranFilter.php
│ ├── JSMinFilter.php
│ ├── JSMinPlusFilter.php
│ ├── LessFilter.php
│ ├── LessphpFilter.php
│ ├── OptiPngFilter.php
```

[1] https://packagist.org/packages/kriswallsmith/assetic.

[2] https://github.com/kriswallsmith/assetic.

```
│ ├── PackagerFilter.php
│ ├── PackerFilter.php
│ ├── PhpCssEmbedFilter.php
│ ├── PngoutFilter.php
│ ├── RooleFilter.php
│ ├── Sass
│ ├── ScssphpFilter.php
│ ├── SprocketsFilter.php
│ ├── StylusFilter.php
│ ├── TypeScriptFilter.php
│ ├── UglifyCssFilter.php
│ ├── UglifyJs2Filter.php
│ ├── UglifyJsFilter.php
│ └── Yui
└── Util
```

На первый взгляд, этот пакет явно нарушает принцип *совместного повторного использования*. Очевидно, что не все классы в этом пакете будут использоваться вместе, поскольку многие фильтры не применяются одновременно одним и тем же пользователем. Возможно, каждому пользователю в действительности нужно всего два или три фильтра.

Но когда мы переключаемся с пользователя на ответственного за поддержку пакета и пытаемся применить принцип *общей закрытости*, следует заметить, что не *все эти классы закрыты* относительно изменений одного и того же типа. Например, если что-то меняется в зависимости от того, как работает компилятор Less, или типа ввода, который ожидает `RooleFilter`, изменение будет сделано только в одном или двух классах внутри пакета. После этого ответственному за поддержку необходимо выпустить новую версию *всего пакета*, чтобы изменения стали доступны всем его пользователям. Для этого потребуется обновить свои проекты (и вероятно, это также внесет множество несвязанных изменений из репозитория), что может или не может привести к нежелательным побочным эффектам.

Чтобы заставить Assetic соответствовать принципу *общей закрытости*, ответственный за поддержку должен создавать отдельные пакеты для каждого фильтра. Каждый пакет для конкретного фильтра будет чувствителен только к конкретным видам изменений (а именно к изменениям, связанным с его собственными зависимостями, такими как компилятор Less). На основной пакет `assetic` такое изменение не повлияет. При изменении одного из специальных фильтров потребуется выпустить только новую вер-

сию пакета для конкретного фильтра. В таких случаях отпадет необходимость выпуска новой версии основного пакета `assetic`.

Следование принципу *общей закрытости* в отношении изменений, внесенных в (необязательные) зависимости, таким образом предотвратит ненужные выпуски пакетов. Это эффективным образом снизит нагрузку на ваших пользователей. Им не нужно обновлять свои зависимости из-за изменений, которые к ним не относятся.

Изменение на уровне приложения

Первое изменение, которое мы обсуждали, было несколько внешним по отношению к пакету: оно было вызвано изменением одной из его зависимостей, будь то пакет, расширение или сам язык программирования. Второй вид изменений связан с тем, что называется *архитектурными уровнями*.

Уровни – это способ применения принципа *единственной ответственности* к приложению. Одним из традиционных методов разбиения приложений на уровни является отделение модели от представления и размещение контроллеров между ними. Контроллеры будут регулировать трафик из контроллера входа вниз в модель и обратно к представлению. В зависимости от того, с кем вам нужно общаться, другие подразделения могут иметь больше смысла. Лично мне нравится использовать в своих приложениях отдельные уровни *Предметная область*, *Приложение* и *Инфраструктура* (если хотите узнать подробности, *см.* мою статью «Уровни, порты и адаптеры», часть 2 «Уровни»[1]). Независимо от того, какие правила вы применяете, разбиение на уровни всегда является полезным организационным шаблоном для приложений.

Большинство приложений уже применяют некую группировку в соответствии с подобластями приложения или набором связанных функций. Получившиеся группы часто называют *модулями*.

Определение модулей следует рассматривать как *вертикальное* разделение, поскольку код для каждого из модулей будет более или менее независимым от других модулей.

Каждый из модулей также можно разделить *горизонтально* путем назначения частей этих модулей их собственным *уровням*. Для

---

[1] https://matthiasnoback.nl/2017/08/layers-ports-and-adapters-part-2-layers/.

этих слоев полезно иметь набор соглашений. Это помогает разработчикам найти подходящее место для каждого файла. В сочетании с правилами о том, как эти уровни могут использовать код друг друга, можно сделать дизайн каждого модуля очень гибким.

Например, помещая всю логику представления в слой *Инфраструктура*, вы позволяете переписывать эту логику, используя другой механизм создания шаблонов, или переключаться с HTML-страниц на ответы в формате JSON без необходимости изменения какого-либо кода в других уровнях (т. е. *Приложение* и *Предметная область*).

Хотя модули приложения обычно не должны заканчиваться пакетами (мы поговорим об этом в конце главы), вы все же можете захотеть повторно использовать модуль (его часть). У вас может возникнуть желание вынуть те файлы, которые вы хотите использовать повторно, и поместить их в один пакет. В конечном итоге это превратит код всех уровней приложения в единый пакет.

Теперь вспомните, что вы ввели разбиение на уровни, чтобы разрешить замену (части) одного из уровней без необходимости изменения каких-либо других уровней. Снова учитывая принцип *общей закрытости*, мы бы не хотели, чтобы изменения вносились только в один уровень пакета. Если это должно произойти, данный принцип побуждает нас разделить пакет, и в этом случае это означает, что мы должны разделить его в соответствии с его уровнями.

Тогда у каждого пакета будет код только для одного уровня, и, следовательно, он будет закрыт относительно изменений одного и того же типа. Это также позволило бы пользователям зависеть от пакетов *Предметная область* и *Приложение* многократно используемого модуля, но реализовывать весь код инфраструктуры самостоятельно.

FOSUserBundle

Давайте рассмотрим пример реального пакета, который использует уровни, – пакет `friendsofsymfony/user-bundle`[1], содержащий `FOSUserBundle`[2]. Его можно использовать в приложении Symfony как способ быстрой настройки пользовательского управления. Будучи в готовом виде, он предоставляет несколько полезных вещей, которые нужны практически каждому веб-приложению:

---

1 https://packagist.org/packages/friendsofsymfony/user-bundle.
2 https://github.com/FriendsOfSymfony/FOSUserBundle/.

- постоянные пользователи и группы пользователей;
- страница сброса пароля;
- страница регистрации;
- страница смены пароля;
- управление пользователями из командной строки
- …

Рассматривая структуру каталогов (*см.* листинг 8-2), можно приблизительно распознать эти функции в имеющихся файлах.

Листинг 8-2. Дерево каталогов пакета friendsofsymfony/user-bundle (сокращенная версия)

```
.
├── Command
│  ├── ActivateUserCommand.php
│  ├── ChangePasswordCommand.php
│  ├── CreateUserCommand.php
│  ├── DeactivateUserCommand.php
│  ├── DemoteUserCommand.php
│  ├── PromoteUserCommand.php
├── Controller
│  ├── ChangePasswordController.php
│  ├── GroupController.php
│  ├── ProfileController.php
│  ├── RegistrationController.php
│  ├── ResettingController.php
├── Doctrine
│  ├── CouchDB
│  ├── MongoDB
│  └── Orm
├── Document
├── Entity
├── Event
├── EventListener
├── Form
├── Mailer
├── Model
├── Propel
└── Resources
├── translations
└── views
```

По совпадению, это еще один пример пакета, который нарушает принцип *совместного повторного использования*, потому что вполне возможно, что кто-то захочет использовать только классы моделей, предоставляемые этим пакетом, в своем собственном

проекте, который не является приложением Symfony. Это невозможно сделать без использования всего пакета `user-bundle`, что в противном случае совершенно бесполезно.

То же самое возражение против конкретного отсутствия связности этого пакета возникает при рассмотрении принципа *общей закрытости* относительно уровней приложений.

Сразу видно, что этот пакет содержит код, связанный со всеми видами уровней.

Следовательно, изменение требований, связанных с пользовательской моделью, приведет к нескольким изменениям в этом пакете. Многие файлы останутся нетронутыми. То же самое касается визуального преобразования веб-страниц, предоставляемых данным пакетом. Чтобы новые шаблоны были доступны всем, нужно выпустить новый релиз, который не имеет отношения к тем, кто использует свои собственные шаблоны. Поскольку в новом выпуске могли измениться некоторые другие вещи, пользователям все равно нужно будет убедиться, что работа их приложения не нарушена, после того как они обновили свои зависимости.

Следовательно, соблюдение принципа *общей закрытости* в отношении уровней приложения потребует от отвечающего за поддержку пакета разбивать этот пакет в соответствии с разными уровнями, чей код и другие ресурсы он содержит. Таким образом, только релевантные изменения заставят пользователей обновлять свои зависимости. В то же самое время это позволяет пользователям заменить одноуровневую реализацию, предоставляемую сопровождающим пакета, своей собственной, без необходимости вставлять неиспользуемый код в свои проекты.

ПРОБЛЕМА, СВЯЗАННАЯ С ПЛАГИНАМИ ДЛЯ ФРЕЙМВОРКОВ

Большинство фреймворков для веб-приложений предлагают комплексное решение для каждого из традиционных уровней приложения (модель, представление и контроллер). Хотя эта идея устарела и многие экспериментируют с другими типами уровней и архитектур, некоторые приложения по-прежнему могут быть сведены к этим основным уровням. Вам всегда нужна какая-то модель предметной области вашего бизнеса. Вам всегда нужен способ показать что-то пользователю своего приложения, и вам всегда нужно что-то, что фреймворк может вызвать, чтобы привести все в движение.

Всякий раз, когда новый фреймворк начинает привлекать внимание разработчиков, все, что уже существует, будет воссоздаваться, чтобы хорошо работать с этим конкретным фреймворком. И поскольку фреймворк обеспечивает стандартный способ работы на каждом уровне, в результате вы получите пакеты:

- в случае с Symfony это «бандлы», которые используют Twig в качестве шаблонизатора и систему объектно-реляционного отображения Doctrine для работы с базами данных;
- в случае с Laravel это «пакеты», которые используют Blade в качестве шаблонизатора и систему объектно-реляционного отображения Eloquent для работы с базами данных;
- в случае с Zend Framework это «модули», которые используют Zend_View в качестве шаблонизатора и Zend_Db для работы с базами данных;
- …

Это не имеет смысла. Это означает, что повторное использование фактически затруднено из-за *ограниченности области повторного использования*. Если люди, ответственные за поддержку этих пакетов, будут больше заботиться о принципах разработки пакетов, они будут делить свои пакеты в соответствии с ответственностями, в отношении зависимостей, а также уровней приложения и тематики. Таким образом, потребуется только один пакет, который моделирует предметную область. Будет несколько пакетов, которые реализуют взаимодействие с базой данных для объектов предметной области, используя различные виды баз данных и соответствующие библиотеки. И также будет несколько пакетов, которые предоставляют уровень представления, работающий с различными шаблонизаторами. Такое разделение пакетов сделает возможным повторное использование (по крайней мере, между проектами, использующими один и тот же язык программирования).

Изменения, продиктованные бизнесом

Мы рассмотрели уровни как способ разделения ваших пакетов. Каждый пакет должен содержать код, связанный только с одним уровнем приложения. Таким образом, когда какое-то требование изменяется в отношении других уровней, пакет останется нетронутым.

Когда вы следуете принципу *общей закрытости* в отношении уровней, то получаете пакеты, которые имеют только одну из ответственностей (например, моделирование, представление и т. д.). Тем не менее эти пакеты будут по-прежнему содержать код, который, вероятно, будет изменен по совершенно другим причинам. Пакет, который содержит весь код уровня предметной области, будет содержать классы, которые моделируют человека, статью, адрес, платеж и т. д. Всякий раз, когда требования уровня предметной области изменяются по причинам, которые диктует бизнес («нам нужно поддерживать еще один тип оплаты»), только один файл в пакете будет изменен, а остальные останутся такими же, какими они были до изменения требований.

Это, конечно, выглядит как еще одно нарушение принципа *общей закрытости*, поскольку классы в пакете должны быть закрыты относительно изменений одного и того же типа. Пакет, полностью предназначенный для уровня предметной области, закрыт относительно всех видов изменений.

Когда вам нужно решить, как группировать классы в пакеты, необходимо рассматривать бизнес-изменения как один из видов изменений, относительно которых классы должны быть закрыты.

Sylius

Позвольте мне закончить этот раздел *хорошим* примером проекта, который разделяет пакеты по предметным областям. Это проект Sylius[1], предлагающий один большой пакет `sylius/sylius`[2], содержащий несколько небольших пакетов (с использованием поддеревьев Git), которые прекрасно разделены по темам (*см.* листинг 8-3).

Листинг 8-3. Пакеты, входящие в состав пакета sylius/sylius

```
SyliusAddressingBundle
SyliusCartBundle
SyliusFlowBundle
SyliusInventoryBundle
SyliusOmnipayBundle
SyliusOrderBundle
SyliusProductBundle
SyliusPromotionsBundle
SyliusResourceBundle
```

1 https://sylius.com.
2 https://packagist.org/packages/sylius/sylius.

```
SyliusSettingsBundle
SyliusShippingBundle
SyliusTaxationBundle
SyliusTaxonomiesBundle
SyliusVariableProductBundle
```

Хотя ни один из этих пакетов не был разделен в соответствии с архитектурными уровнями, когда дело касается проблемной области, все эти пакеты прекрасно отделяются друг от друга. Таким образом, человек, отвечающий за поддержку, сможет ограничить необходимые изменения, изменяя требования только к одному или двум пакетам одновременно.

Бизнес-логика

В этой главе мы рассмотрели различные темы из области разработки приложений (в отличие от разработки *пакетов*). Мы говорили об архитектурной иерархии – о том, что обычно рассматривается в контексте приложения, а не пакета. Мы также говорили о логике предметной области, или бизнес-логике, и почему следует ограничить пакет до конкретной подобласти. Опять же, это разделение на подобласти обычно рассматривается только в рамках более крупного программного проекта и не так часто в контексте разработки пакетов.

Причина состоит в том, что использование архитектурных уровней и разделение предметной области на подобласти обычно необходимы только при наличии достаточного количества кода. Поскольку три принципа связности в целом приведут к появлению пакетов меньшего размера, останется не так много кода, который нужно будет разбивать. Поэтому вы обнаружите, что большинство пакетов принадлежат только одному уровню и даже не охватывают несколько подобластей. Давайте посмотрим на общую картину, чтобы узнать, почему это происходит.

Каждому приложению нужна система управления пользователями, страница входа в систему, страница сброса пароля, некий способ отслеживания прав пользователя, модули управления, где администраторы могут быстро войти в базу данных и что-то изменить для клиента и т. д. Будучи программистом, иногда у вас может возникнуть ощущение, что вы начинаете понимать суть вопроса только спустя несколько недель разработки таких технических основ для нового приложения.

Это означает, что вы интуитивно знаете, какая часть общей предметной области занимает центральное место в приложении, над которым вы сейчас работаете, а какие менее значимы.

Конечно, система авторизации должна быть. Она должна функционировать правильно и быть надежной. Но система авторизации – это то, что нужно многим другим командам, и ее не нужно изобретать заново. Вместо этого было бы разумно покончить с этой частью вашего приложения как можно скорее и начать работать над частями, где ваше приложение может иметь реальное значение.

Эрик Эванс так резюмирует свой совет по этой теме:

> *Определите связанные подобласти, которые не являются мотивацией для вашего проекта. Выделите из них неспециализированные модели и поместите их в отдельные модули. Не оставляйте в них следов своих специальных приложений. Как только они будут разделены, присвойте их непрерывной разработке более низкий приоритет по сравнению со смысловым ядром и не назначайте своих главных разработчиков на эти задачи (потому что они получат от них мало знаний предметной области). Также рассмотрите готовые решения или опубликованные модели для этих неспециализированных подобластей*[1].

Он называет это *стратегической дистилляцией* – выяснить, что является вашим смысловым ядром, и, таким образом, распознать подобласти, в том числе «неспециализированные». Для разработчиков *приложений* это полезный совет, поскольку он помогает сосредоточить усилия по разработке на тех областях, где ваше приложение может выделяться среди множества других. В то же время это поможет вам решить, для каких частей вашего приложения лучше использовать существующую библиотеку или внешнюю службу, что также известно как «готовое решение».

Для разработчиков *пакетов* это также полезный совет, поскольку разработчики приложений, которые ищут готовые решения, могут быть *пользователями* пакетов, которые вы публикуете. Поэтому, когда вы подумаете о том, чтобы извлечь часть вашего приложения в пакет, который можно использовать многократно, подумайте, могут ли его использовать другие, чтобы помочь им быстрее добраться до своего смыслового ядра. `FOSUserBundle` явля-

---

[1] *Eric Evans*. Domain-Driven Design. Addison-Wesley Professional (2003).

ется отличным примером такого пакета, который выглядит так, как будто он был извлечен из проекта, чтобы его можно было использовать снова и снова во всех проектах, для которых требуется управление пользователями и система авторизации.

Вы также можете создать более крупный пакет повторного использования или набор пакетов и включить в него полноценную модель предметной области и сервисы приложений (как это делает Sylius для программного обеспечения электронной коммерции). Сделать такие пакеты полезными для всех пользователей будет непросто, потому что код в нем должен быть специфичным для некоторого понимания предметной области. Это означает, что пользователи захотят изменить встроенное поведение. Однако как только это произойдет, пользователям также придется оценить, правильно ли они разделили смысловое ядро и неспециализированные подобласти. Повторно используемый код в основном успешен для неспециализированных подобластей; смысловое ядро нуждается в пользовательских усилиях по моделированию, чтобы все было сделано правильно. И чтобы ему было легче развиваться в соответствии с меняющимся требованиями бизнеса.

Поскольку классы, которые вы пишете для подобластей, отличных от неспециализированных, скорее всего, будут очень специфичными и бесполезными в других проектах, не кладите их в пакеты. Просто используйте пространства имен для группировки классов в подобластях, к которым они принадлежат. Точно так же вы можете использовать пространства имен для структурирования кода приложения в соответствии с архитектурными уровнями, которые вы хотите применить. Это избавит вас от множества разочарований, когда вы будете работать над приложением. Изменение в приложении можно выпустить за один раз, даже если оно охватывает несколько уровней и подобластей. Если вместо этого вы будете создавать пакеты для каждой комбинации архитектурного уровня и подобласти, то вскоре получите неподдерживаемое приложение, которое очень устойчиво к изменениям.

Треугольник принципов связности

Прежде чем мы перейдем к следующему набору принципов разработки пакетов, связанных с зацеплением, мы кратко обсудим интересную концепцию, упомянутую Робертом Мартином в одном из

его обучающих видео на сайте https://cleancoders.com[1]. Она называется «треугольником принципов связности» и показана на рис. 8-1.

Рис. 8-1. Треугольник принципов связности

В каждом углу одного из принципов связности пакетов можно нарисовать треугольник. Затем можно расположить любой пакет где-нибудь в этом треугольнике. Перемещение пакета в один из углов означает, что он максимально реализует этот принцип, игнорируя другие принципы. Перемещение к вершине, противоположной одному из углов, означает, что этот пакет определенно не следует соответствующему принципу, но одинаково хорошо следует другим принципам.

Роберт предполагает, что пакет может перемещаться внутри диаграммы. Когда вы впервые начинаете работать над проектом, пакеты в первую очередь можно разрабатывать в соответствии с принципом *совместного повторного использования*. В дальнейшем вы можете более строго придерживаться принципа *общей закрытости*, поскольку это облегчит поддержку. Затем, в конце, вы можете сосредоточиться на том, чтобы сделать пакет повторно используемым, чтобы все усилия, которые вы вложили в него, давали вам преимущество в вашем следующем проекте (или помогали другим разработчикам по всему миру решать проблемы, которые вы уже решили).

Этот треугольник – хороший способ оценить качество связности пакета в любой момент времени. Легко заметить, когда пакет не следует принципу *эквивалентности повторного использования и выпуска*, потому что тогда его трудно добавить в качестве зави-

---

[1] https://cleancoders.com/episode/clean-code-episode-16/show.

симости в свой проект и трудно поддерживать его в актуальном состоянии, как только вам это удастся. Также легко проверить, насколько хорошо соблюдается принцип *совместного повторного использования*: когда вы чувствуете, что вам нужно получить большое количество кода, просто чтобы использовать один или два класса, тогда что-то пошло не так. И наконец, если пакет содержит классы, которые имеют ответственности во множестве разных областей знаний, принцип *общей закрытости* не очень хорошо соблюдается. Нарисуйте треугольник для одного из своих пакетов, и это может помочь вам узнать, как улучшить его.

Заключение

Когда какой-либо аспект пакета должен измениться, он должен быть «открыт». Код должен быть изменен, и обновленный пакет должен быть выпущен снова. Тому, кто отвечает за поддержку пакета, будет легче, если его не нужно будет модифицировать и выпускать снова. Следовательно, принцип общей закрытости говорит нам о том, что необходимо рассмотреть все причины, по которым пакет может потребоваться изменить, и соответствующим образом разделить его. Это в конечном итоге сделает каждый пакет «закрытым относительно изменений одного и того же типа».

Распространенные причины открытия пакета:

- если что-то меняется в зависимости (например, нужно что-то модернизировать или заменить);
- если требования относительно какой-то части бизнес-логики изменились;
- если часть инфраструктуры изменяется, но основная логика остается неизменной (например, при изменении макета пользовательского интерфейса или при переключении на другую базу данных).

Если занимаетесь поддержкой пакета, следите за тем, сколько пакетов необходимо выпустить снова после изменения. Ваша цель – минимизировать это число, разделив пакет. Также следите за тем, сколько классов изменено для каждого выпуска. Если это только часть всех классов в этом пакете, разделите пакет. Перемещение классов, которые редко изменяются или изменяются по разным причинам, облегчит обслуживание в будущем. Это будет полезно и для пользователей, поскольку им не нужно будет обновлять пакет по причинам, которые не имеют к ним отношения.

Глава 9

Принцип ацикличности зависимостей

Как я объяснил во введении, у всех программистов возникает чувство «связи друг с другом». Но помимо этой интуиции относительно связности, у программистов также есть нюх на зацепление. Глядя на фрагмент кода, они смогут выяснить, от чего он зависит. По мере продвижения по карьерной лестнице они будут разрабатывать все более сильные «радары» для обнаружения зацепления, выясняя фактические *зависимости* любого фрагмента кода.

Зацепление: выявление зависимостей

Глядя на некий код, вы можете спросить себя: на *что* еще полагается этот код для успешного выполнения? Если долго и усердно обдумывать данный вопрос, обнаруживаются некоторые очевидные зависимости, но, вероятно, также и некоторые менее очевидные, или косвенные, зависимости.

Рассмотрим класс `Kernel`, определенный в листинге 9-1.

Листинг 9-1. Класс Kernel

```
namespace SomeFramework;
class Kernel
{
public function __construct(EventDispatcher $eventDispatcher)
{
// ...
}
}
```

Чтобы успешно использоваться в приложении, класс `Kernel` зависит от:

- класса `EventDispatcher`. Он получает экземпляр этого класса в качестве аргумента конструктора;
- интерпретатора PHP. Это нужно для того, чтобы вообще запуститься. В частности, версия интерпретатора должна быть не ниже 5.3, поскольку класс находится в пространстве имен, которое не поддерживается более ранними версиями PHP.

Мы могли бы пойти намного дальше в определении зависимостей. Интерпретатор PHP зависит от операционной системы, работающей на компьютере, который нуждается в энергии и должен быть доступен вам как пользователю данного программного обеспечения. Поэтому он должен находиться в этом мире, в этой вселенной (ну, может быть, ваш код совместим даже со вселенной, кто знает?) и т. д.

Я согласен с вами, что так можно зайти слишком далеко. Однако простого просмотра классов, используемых в коде, и версии PHP, необходимой для выполнения кода, в большинстве случаев недостаточно. Код может зависеть от того, что сервер базы данных запущен и работает и доступен с сервера, на котором выполняется код. Или, возможно, для выполнения кода требуется определенный объем памяти, или пользователь, который запускает программное обеспечение, должен иметь определенные права доступа к файловой системе и т. д.

В этой главе и в последующих двух главах мы будем обсуждать зависимости пакетов. Когда мы будем обсуждать принципы зацепления пакетов, мы не будем рассматривать физические зависимости пакета. Мы будем принимать во внимание только другие единицы кода – внешние по отношению к пакету. Эти *внешние единицы кода* могут быть фактическими пакетами (с файлом определений или без него), языковые расширения (которые на самом деле представляют собой особый вид самих пакетов) или другие скопления кода, необходимые для запуска кода в данном пакете.

Различные способы зацепления пакетов

Давайте сначала остановимся на следующем соглашении: мы называем «данный пакет» «корневым пакетом». Начиная с любого корневого пакета, мы можем перечислить зависимости этого пакета. Этими зависимостями могут быть любые пакеты или даже языковое расширение, но мы просто называем эти зависимости «пакетами», чтобы было проще.

Теперь мы можем составить список способов, с помощью которых код в корневом пакете вводит зависимости от других пакетов.

Поскольку мы отвечаем за поддержку пакета, нам понадобится этот список, когда мы будем собирать имена необходимых пакетов для файла определений. Нам также понадобится этот список зависимостей, когда мы будем применять принципы зацепления пакетов. Когда мы не знаем точно, каким образом код в корневом пакете использует зацепление с другими пакетами, мы не будем знать, как нарисовать граф зависимостей корневого пакета как часть крупной системы пакетов. Мы также не будем знать, как исправить проблемы в графе зависимостей.

Композиция

Мы уже рассматривали один конкретный способ зацепления: когда тип аргумента конструктора является классом (*см.* листинг 9-2).

Листинг 9-2. Зацепление посредством аргументов конструктора

```
namespace RootPackage;
use OtherPackage\EventDispatcher;
class Kernel
{
private $eventDispatcher;
public function __construct(EventDispatcher $eventDispatcher)
{
$this->eventDispatcher = $eventDispatcher;
}
}
```

Класс `Kernel` находится в корневом пакете. Когда класс `EventDispatcher` находится в другом пакете (как и должно быть), этот конкретный фрагмент кода использует зацепление пакетов. Например, пакет `kernel` будет зависеть от пакета `event-dispatcher`. Это называется зависимостью *по композиции*, поскольку шаблон хранения одного объекта внутри другого объекта для последующего использования называется «композицией».

Наследование

Еще одним способом, с помощью которого класс может быть связан с другим классом, является наследование (*см.* листинг 9-3).

Листинг 9-3. Зацепление посредством наследования

```
namespace RootPackage;
use OtherPackage\Controller;
class LoginController extends Controller
```

```
{
// ...
}
```

Когда родительский класс `Controller` (или любой из его родительских классов) находится в другом пакете, а не в `LoginController`, наследование использует зацепление пакетов.

Реализация

Очень похоже на наследование, реализация интерфейса или расширение абстрактного класса и реализация его абстрактных методов также использует зацепление (*см.* листинг 9-4).

Листинг 9-4. Зацепление посредством реализации интерфейса

```
namespace RootPackage;
use OtherPackage\RequestListener;
class IpBlocker implements RequestListener
{
// ...
}
```

Использование

Часто зацепление между классами происходит, когда экземпляр одного класса просто *использует* экземпляр другого класса, например в качестве одного из параметров метода (*см.* листинг 9-5).

Листинг 9-5. Зацепление посредством аргументов метода

```
namespace RootPackage;
use OtherPackage\NewRequestEvent;
class IpBlocker
{
public function onKernelRequest(NewRequestEvent $event): void
{
// ...
}
}
```

Инстанцирование

В предыдущих примерах зависимости от классов вне корневого пакета оставались открытыми, поскольку они были частью открытого интерфейса (родительский класс, реализованный интерфейс и тип аргумента метода). Но существует еще и несколько закрытых способов зацепления. Например, когда один объект создает другие объекты класса внутри другого пакета (*см.* листинг 9-6).

Листинг 9-6. Зацепление посредством инстанцирования объекта

```
namespace RootPackage;
use OtherPackage\ServiceContainer;
class Kernel
{
public function boot(): void
{
$container = new ServiceContainer();
// ...
}
}
```

Использование глобальной функции

Обычными подозреваемыми для зацепления являются классы, но стоит приглядеться и к функциям, которые используются в пакете. Многие функции доступны только при установке определенного пакета или языкового расширения, например расширения `curl` (*см.* листинг 9-7).

Листинг 9-7. Зацепление посредством использования функции

```
class HttpClient
{
public function send(): void
{
$ch = curl_init();
// ...
}
}
```

Функции используются только во внутренних частях класса, т. е. в теле его методов. Таким образом, обнаружение этих зависимостей требует немного больше усилий. То же самое касается общедоступных статических методов, которые также вводят зацепление внутри методов, как показано в листинге 9-8.

Листинг 9-8. Зацепление посредством использования общедоступного статического метода

```
class Controller
{
public function indexAction()
{
$translator = Zend_Registry::get('Zend_Translator');
// ...
}
}
```

Что не следует учитывать: глобальное состояние

Код в пакете часто зависит от определенного глобального состояния. Наиболее очевидный пример – каждый пакет неявно зависит от наличия автозагрузчика, который способен (автоматически) загружать классы внутри пакета.

В противном случае всегда следует избегать зависимости от глобального состояния, но даже когда это происходит, в этой книге мы не рассматриваем это как зависимость пакета, поскольку не можем сделать ее явной в списке требований пакета.

Визуализация зависимостей

Прежде чем мы сможем приступить к применению принципов зацепления для разработки своих пакетов, мы должны иметь возможность визуализировать любое текущее состояние зацепления. Делая это, мы одновременно рассматриваем только одну систему (то есть приложение). Мы берем все пакеты, которые находятся внутри этой системы, и рассматриваем их один за другим как корневой пакет. Затем мы используем этот список типов зацеплений (например, композиция, наследование и т. д.) для извлечения списка всех зависимостей классов в этом пакете. Как только такая зависимость класса выходит за пределы самого пакета, она должна быть помечена как *зависимость пакета*.

Результатом такого упражнения будет следующий список:

- пакет A зависит от пакета C;
- пакет B зависит от пакетов A и D;
- пакет C зависит от пакета B.

Пакеты и их зависимости, когда они записываются, как показано на рис. 9-1, образуют рецепт графа, где каждый пакет – это *вершина* (узел), а каждая зависимость – это *ребро* (линия).

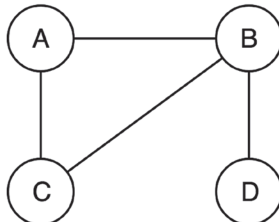


Рис. 9-1. Граф с вершинами в виде пакетов и ребрами в качестве зависимостей

Поскольку зависимости имеют направленность (пакет зависит от другого пакета, а не наоборот), мы можем преобразовать этот граф в ориентированный, просто добавив несколько стрелок к ребрам, как показано на рис. 9-2.

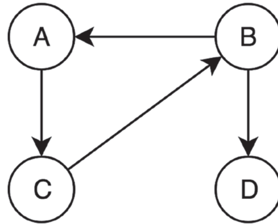


Рис. 9-2. Ориентированный граф

Говоря об этих диаграммах, можно упомянуть автореферентные пакеты. На данный момент они не представляют интереса, хотя я кратко вернусь к этой теме позже. Практически все пакеты являются автореферентными, потому что часто класс в корневом пакете использует другой класс или функцию из того же пакета.

Когда дело касается зависимостей пакетов, есть еще один важный аспект, который мы должны учитывать при построении графа зависимостей пакетов в системе: возможно, существуют некоторые ограничения версий в отношении зависимостей.

Например, пакету C может потребоваться как минимум версия 1.0 пакета B. Мы можем записать эти ограничения в виде аннотаций в графе, как показано на рис. 9-3.

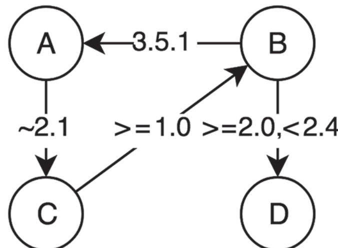


Рис. 9-3. Ориентированный граф с аннотациями, обозначающими ограничения версий

Принцип ацикличности зависимостей

Все эти полученные знания о зацеплении пакетов – лишь один из шажков к объяснению принципа *ацикличности зависимостей*, который гласит:

> *Структура зависимостей между пакетами должна быть ориентированным ациклическим графом; то есть в структуре зависимостей не должно быть циклов*[1].

Мы уже обсуждали, как можно выявить фактические зависимости всех пакетов в системе, а затем нарисовать *ориентированный граф* результатов. Единственный недостающий фрагмент информации – что такое *ациклический* ориентированный граф.

Ориентированный граф не имеет циклов, если, начиная с любой вершины, нет пути, который через любое количество вершин ведет обратно к исходной вершине. Такой ориентированный граф называется *ациклическим*. Если перевести это на язык графов зависимостей – какой бы пакет вы ни выбрали в качестве корневого, следуя стрелкам зависимостей, вы не сможете вернуться к корневому пакету. *См.* рис. 9-4.

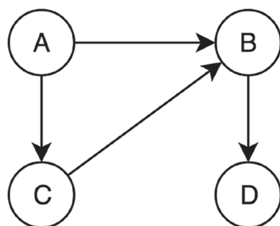


Рис. 9-4. Ациклический ориентированный граф: циклы отсутствуют

И наоборот, если в ориентированном графе *есть* циклы, это означает, что есть *хотя бы одна* вершина, для которой можно найти путь, ведущий обратно к той же самой вершине. Говоря терминами графа зависимостей, это означает, что есть, по крайней мере, один пакет, который является началом пути последующих зависимостей, который ведет к тому же пакету (*см.* рис. 9-5).

---

[1] *Robert C. Martin.* Engineering Notebook. C++ Report, Nov-Dec, 1996 (PDF-версия доступна по адресу: http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod).

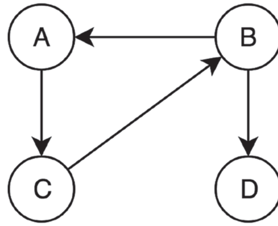


Рис. 9-5. Циклический ориентированный граф с одним циклом

Принцип *ацикличности зависимостей* гласит, что когда вы рисуете свой граф зависимостей, он должен выглядеть как *ациклический ориентированный граф*, а это означает, что у него нет циклов.

Проблемные циклы

Будучи программистом, вы, вероятно, знакомы с циклами, к которым часто приводят легко совершаемые ошибки, как те, что показаны ниже (*см.* листинг 9-9).

Листинг 9-9. Ошибки в программировании, приводящие к появлению циклов

```
$somethingThatIsAlwaysTrue = ...;
while ($somethingThatIsAlwaysTrue) {
// ...
}
// или нечто наподобие:
for ($i = 0; $i < 100;) {
//Мы забыли о приращении...
}
// а также:
class Node
{
private $parent;
public function getParent(): ?Node
{
if ($this->parent) {
return $this->parent;
}
// Гм!
return $this->getParent();
}
}
```

Пример распространенного типа цикла, который на самом деле является не ошибкой, а скорее проблемой проектирования, приведен в листинге 9-10.

Листинг 9-10. Классам Desk и Programmer нужны экземпляры друг друга

```
class Desk
{
private $programmer;
public function __construct(Programmer $programmer)
{
$this->programmer = $programmer;
}
}
class Programmer
{
private $desk;
public function __construct(Desk $desk)
{
$this->desk = $desk;
}
}
$desk = new Desk($programmer);
$programmer = new Programmer($desk);
// ??
```

Как бы мы ни старались, у нас ничего не получится. Это настоящая циклическая зависимость. Сначала нам нужно создать экземпляр `Desk`, но для этого нам нужен `Programmer`, которому нужен `Desk`, и т. д.

Обычно мы решаем подобные проблемы, сначала создавая объект, а затем внедряя зависимость, которая вызвала цикл (*см.* листинг 9-11).

Листинг 9-11. Разрываем цикл

```
class Desk
{
private $programmer;
public function setProgrammer(Programmer $programmer): void
{
$this->programmer = $programmer;
}
}
class Programmer
{
private $desk;
public function __construct(Desk $desk)
```

```
{
$desk->setProgrammer($this);
$this->desk = $desk;
}
}
$desk = new Desk();
$programmer = new Programmer($desk);
```

Используя это решение, вы можете быть уверены, что в какой-то момент *хотя бы один* из объектов находится в недопустимом состоянии. В данном случае это объект Desk. У него нет ассоциированного Programmer, пока он не будет назначен в конструкторе самого Programmer, поэтому это не окончательное решение.

Вместо того чтобы жертвовать согласованным состоянием наших объектов, мы должны найти способ разорвать цикл. В реальных ситуациях моделирования вы можете задать следующие вопросы, чтобы найти более подходящее решение:

- вам действительно нужен весь объект или только его часть, может быть, даже одно значение? Если вам нужна только часть объекта, вы можете внедрить эту часть и эффективно удалить цикл;
- эти два объекта действительно должны знать друг о друге? Можно ли изменить отношение с двунаправленного на однонаправленное?

Проблемы, описанные ранее, обычно не выходят за границы пакета, а остаются приватными для пакета и должны быть устранены там. Однако когда циклическая зависимость между классами выходит за границы содержащего их пакета, она становится *круговой зависимостью между пакетами*, а затем возникают другие, более серьезные проблемы.

Для некоторых диспетчеров пакетов или зависимостей может быть невозможно разрешить циклические зависимости. То же самое касается и внедрения зависимостей, например контейнеров служб, у которых может не быть возможности инстанцировать службы, если их зависимости тоже образуют цикл (например, если служба B зависит от службы A, но в конечном итоге оказывается, что инстанцирование службы A зависит от правильно инстанцированнной службы B).

Но даже если разрешение циклических зависимостей не является проблемой для выбранного вами диспетчера пакетов, управление выпусками циклических зависимостей по-прежнему может быть проблематичным. Представьте себе, что выпуск следующей старшей

версии пакета A будет зависеть от следующей старшей версии пакета B, которая переписывается, чтобы в полной мере воспользоваться всеми преимуществами, которые будут выпущены в следующей старшей версии пакета A. Какой пакет должен быть выпущен первым?

Будет сложно координировать старшие релизы, когда граф зависимостей имеет циклы, хотя это можно преодолеть, сделав релизы-«камикадзе», добавив некоторые меры обратной или прямой совместимости и предоставив дружественные ограничения версий для соответствующих зависимостей пакетов. В конце концов, возможно, это и не причинит вам слишком много неприятностей. Проблема циклических зависимостей пакетов на самом деле намного больше, когда речь идет о языках программирования, где есть процесс сборки. Когда разрешаются зависимости процесса сборки, циклическая зависимость может даже помешать успешному завершению всего процесса сборки.

Тем не менее существует интуиция программиста, которая говорит, что с циклами в программном обеспечении что-то не так, в частности с циклами, которые выходят за пределы одного пакета. Поэтому в любом случае их можно исправить. Хорошая новость состоит в том, что существует несколько простых решений для удаления циклов из графа зависимостей.

Решения, позволяющие разорвать циклы

Во-первых, некоторые циклы «более реальны», чем другие.

Псевдоциклы и избавление от них

Рассмотрим два пакета, показанных на рис. 9-6, которые имеют прямую зависимость друг от друга (для этого примера не имеет значения, состоит ли цикл из большего числа пакетов).

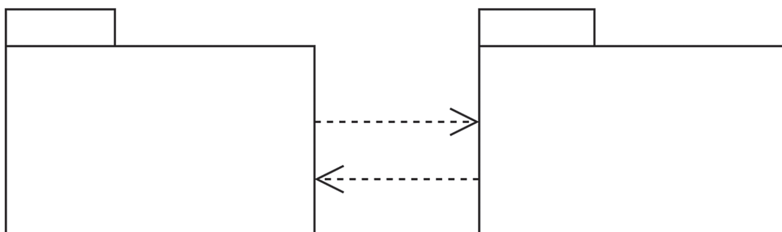


Рис. 9-6. Два пакета, которые зависят друг от друга

Когда зависимость считается «пакетной», это означает, что часть кода в *корневом* пакете зависит от части кода в *другом* пакете. Чаще всего корневой пакет содержит класс, который использует класс из другого пакета одним из способов, описанных в начале этой главы. При внимательном рассмотрении обоих пакетов мы обнаруживаем, что в этом случае, и правда, один класс в корневом пакете (класс А) зависит от класса в другом пакете (класс С), что объясняет одно направление пакетной зависимости (*см.* рис. 9-7).

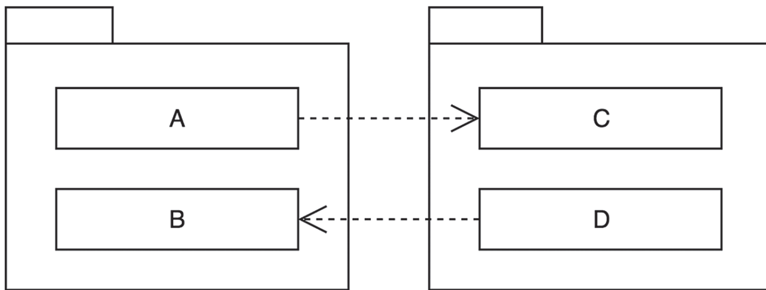


Рис. 9-7. Если внимательно посмотреть на фактические зависимости

Пытаясь объяснить причину, по которой эта зависимость пакета является *циклической*, мы отмечаем, что взаимная зависимость совершенно не нужна, поскольку она касается двух других, несвязанных классов – классов B и D.

Это означает, что эти два пакета используются двумя разными, не связанными между собой способами и что классы не были правильно разделены между ними. Эти пакеты просто нарушают принцип *совместного повторного использования*: не все их классы используются повторно одновременно. Это также напоминает принцип *разделения интерфейса* (глава 4), но применяется к самому пакету, а не только к одному классу.

Очевидно, для этого пакета существует несколько разных *клиентов*, и только некоторые из них вызывают цикл зависимости.

Я называю такой тип цикла зависимости *псевдоциклом*. От него можно легко избавиться, перегруппировав код и создав один или два новых пакета. Например, можно поместить классы B и D в один пакет, что делает зависимость внутренней для этого пакета, как показано на рис. 9-8.

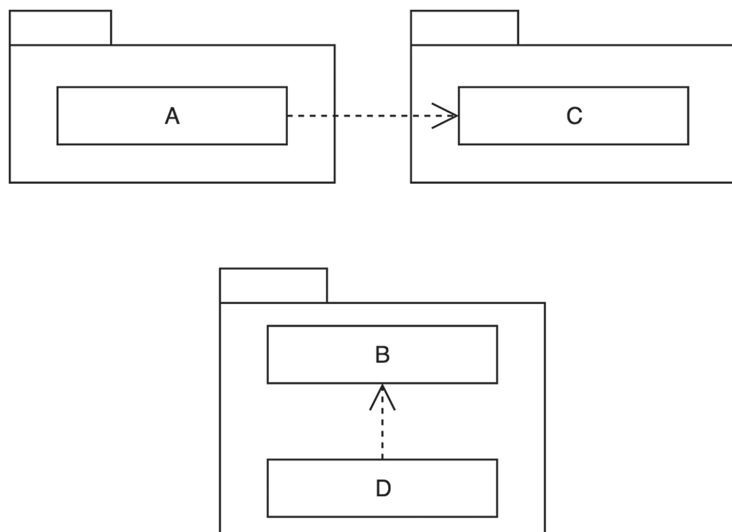


Рис. 9-8. Одно из решений, позволяющее избавиться от псевдоцикла

Либо можно поместить классы B и D в их отдельные пакеты, в результате чего пакетная зависимость останется как есть, но, по крайней мере, цикл из графа зависимостей будет удален (*см.* рис. 9-9).

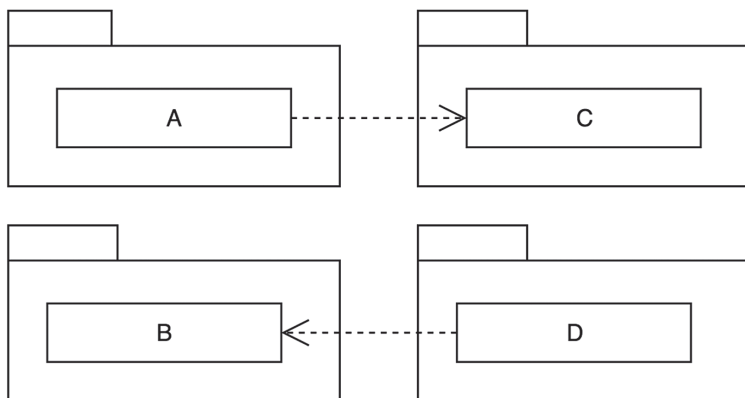


Рис. 9-9. Еще одно решение, позволяющее избавиться от псевдоцикла

Вам следует выбрать второе решение, когда другие части системы полагаются только на класс B или только на класс D, а это означает, что согласно принципу *совместного повторного использования* эти классы должны находиться в отдельных пакетах.

Реальные циклы и избавление от них

Мы только что обсудили прекрасный псевдоцикл, но как выглядит реальный цикл? На рис. 9-10 показан схематический пример такой циклической зависимости.

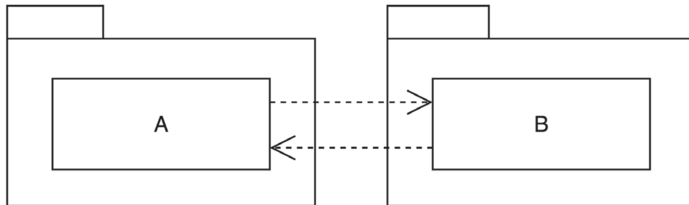


Рис. 9-10. Реальный цикл

Опять же, между этими двумя классами может быть любое количество пакетов, пока существует фактический цикл.

Чтобы внести больше конкретики, предположим, что класс A используется для обработки веб-форм, а класс B является обычным валидатором. Формы и валидация очень тесно связаны, но валидация, по крайней мере, не ограничивается формами. Таким образом, пакет валидации должен быть доступен для отдельного использования в различных сценариях (например, для проверки параметров запроса или десериализованных объектов).

В пакете форм мы находим класс `Form` (*см.* листинг 9-12).

Листинг 9-12. Класс Form из пакета Form

```
use Validator\Validator;
class Form
{
private $validator;
private $errors = [];
public function __construct(Validator $validator)
{
$this->validator = $validator;
}
public function isValid(): bool
```

```
{
$this->validator->validateForm($this);
return count($this->errors) === 0;
}
public function addError(Error $error): void
{
$this->errors[] = $error;
}
}
```

В пакете валидатора мы находим класс `Validator` (*см.* листинг 9-13).

Листинг 9-13. Класс Validator из пакета Validator

```
use Form\Form;
class Validator
{
public function validateForm(Form $form): void
{
// ...
$form->addError(...);
}
}
```

Эти фрагменты кода раскрывают циклическую зависимость между `Validator` и `Form` типа «использование» (*см.* список типов зацепления в начале этой главы).

Чтобы разорвать этот тип цикла, недостаточно просто переместить код (как это было, когда мы обсуждали псевдоцикл в предыдущем разделе). Мы должны заняться программированием, чтобы решить эту проблему. Мы проведем рефакторинг кода, то есть изменим его структуру, а не поведение, применив к нему некоторые шаблоны проектирования. Тут есть много возможных решений, и я покажу некоторые наиболее часто используемые.

Инверсия зависимостей

Первое, что мы можем сделать, – это удалить жесткую зависимость от класса в другом пакете. Как объяснялось в главе 5, применяя принцип *инверсии зависимостей*, можно легко изменить направления зависимости, завися от чего-то *абстрактного*, то есть интерфейса, а не чего-то *конкретного*, то есть класса. Если мы затем переместим интерфейсы в отдельные пакеты, то будем спасены (*см.* листинг 9-14).

Листинг 9-14. Интерфейсы FormInterface и ValidatorInterface

```
interface FormInterface
{
public function isValid(): bool;
public function addError(Error $error): void;
}
class Form implements FormInterface
{
public function __construct(ValidatorInterface $validator)
{
// ...
}
public function isValid(): bool
{
// ...
}
public function addError(Error $error)
{
// ...
}
}
interface ValidatorInterface
{
public function validateForm(FormInterface $form): void;
}
class Validator implements ValidatorInterface
{
public function validateForm(FormInterface $form): void
{
// ...
}
}
```

Мы вводим интерфейсы и заставляем существующие классы реализовывать их. Потом мы используем только интерфейсы в качестве типов параметров, а не классы. Когда мы помещаем интерфейсы в отдельные пакеты, мы успешно отклоняем некоторые проблемные зависимости, и на графике зависимостей не остается никаких кружков, как показано на рис. 9-11.

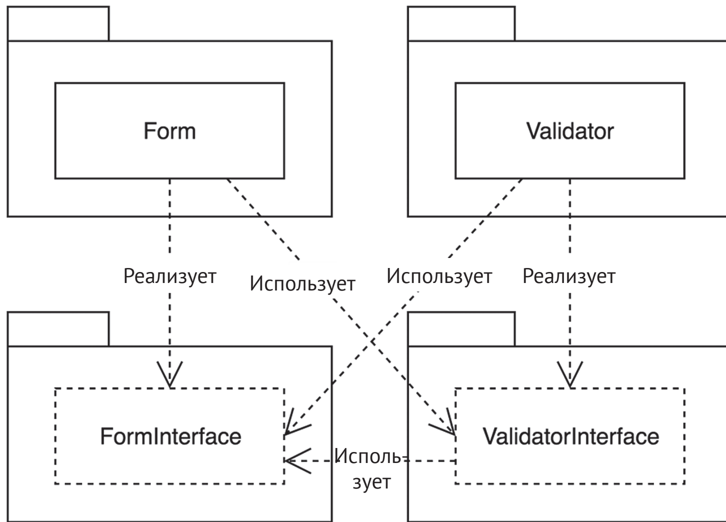


Рис. 9-11. Удаление цикла с помощью инверсии зависимостей

Хитрость здесь состоит в том, что хотя `Form` изначально зависел от экземпляра `Validator` (внедренного в качестве аргумента конструктора), а `FormInterface` нет, это означает, что стрелка зависимости, ведущая от `FormInterface` к `ValidatorInterface`, отсутствует.

Итак, используя инверсию зависимостей, мы фактически «растворили» цикл в графе зависимостей. Однако это не самое лучшее решение, поскольку нам пришлось переместить `Form` в его собственный пакет.

Инверсия управления

Проблема проектирования классов `Form` и `Validator`, которая привела к их тесной связи, состоит в том, что они слишком много знают друг о друге, и это приводит к большому общению между ними. Помните, что причиной наличия отдельного пакета для проверки данных было то, что проверка данных не обязательно ограничена проверкой данных, представленных с помощью веб-формы. Поэтому, по меньшей мере, удивительно, что класс `Validator` действительно содержит метод `validateForm()`. Это явно нарушает принцип *разделения интерфейса* (глава 4), поскольку только часть *клиентов* класса `Validator` будет использовать этот метод.

Когда объект обменивается данными с другим объектом, вызывая методы для него, он действительно осуществляет контроль над ним. Вызов метода инициирует действие в другом объекте. Когда объекты вызывают методы друг друга, то есть *общаются друг с другом*, это следует рассматривать как *осуществление контроля* друг над другом. Но общение, которое идет вперед и назад, создает цикл. Чтобы разрешить этот цикл, мы должны разорвать линии передачи данных между объектами и позволить вести разговор некоему промежуточному объекту. Это, по сути, *инвертирует* направление контролирующего поведения данных объектов. Следовательно, этот метод известен как *инверсия управления*.

Есть много вариантов, когда вы хотите выполнить рефакторинг кода, который слишком уж держит все «под контролем». Фактически все шаблоны проектирования, известные как «поведенческие шаблоны» (*см.* также знаменитую книгу «Банды четырех» под названием «*Шаблоны проектирования: элементы многократно используемого объектно-ориентированного программного обеспечения*») подходят для этой цели. Вам решать, применимы ли они к вашей ситуации. Кроме того, не обязательно следовать точным шаблонам.

Посредник

Первое и самое простое решение – использовать шаблон *Посредник*[1]. Тогда объект `Form` больше не должен делать никаких прямых вызовов объекта `Validator`. Вместо этого он может вызывать только посредника, который, в свою очередь, будет выполнять любые специфичные для формы вызовы `Validator` (*см.* листинг 9-15).

Листинг 9-15. Класс FormValidationMediator

```
class FormValidationMediator
{
private $validator;
public function __construct(ValidatorInterface $validator)
{
$this->validator = $validator;
}
public function validate(FormInterface $form): void
{
$formData = $form->getData();
```

[1] *Erich Gamma e. a.* Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

```
$errors = $this->validator->validate($formData);
foreach ($errors as $error) {
$form->addError($error);
}
}
}
class Form implements FormInterface
{
private $formValidator;
public function __construct(
FormValidationMediator $formValidator
) {
$this->formValidator = $formValidator;
}
public function isValid(): bool
{
//FormValidationMediator добавит ошибки в форму;
$this->formValidator->validate($this);
return count($this->errors) === 0;
}
}
```

По сути, зависимость из пакета валидатора удаляется в пакет формы. `Validator` теперь действительно автономен. Посредник, который выполняет проверку в формах, должен находиться в собственном пакете, с зависимостями как от пакетов форм, так и от интерфейсов валидатора, как показано на рис. 9-12.

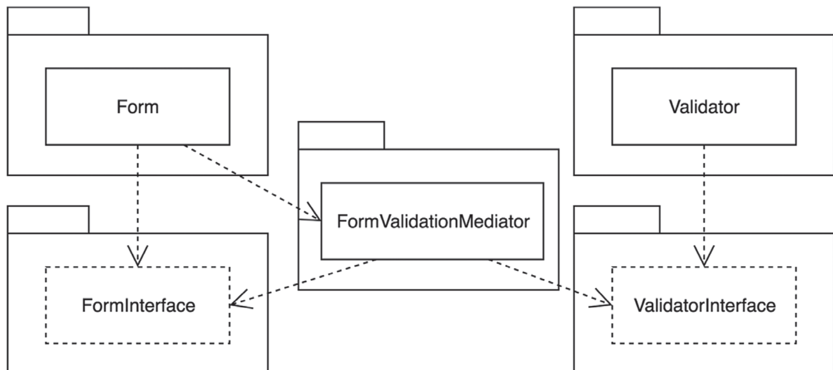


Рис. 9-12. Перемещение зависимости из ValidatorInterface в FormInterface путем введения еще одного пакета, который будет посредником между ними

ИМЕНОВАНИЕ КЛАССОВ, ВДОХНОВЛЕННЫХ ШАБЛОНАМИ

Я думаю, что `FormValidationMediator` – немного странное имя для класса, поскольку оно включает в себя имя использованного шаблона. По моему опыту более разумно будет дать классу имя, которое наиболее соответствует контексту вашего приложения, и убедиться, что оно читабельно.

Затем в документации класса можно упомянуть шаблон, который вы применяли, и, возможно, причину, по которой вы его использовали, если это может помочь читателю понять, что происходит (как показано в листинге 9-16).

Листинг 9-16. Использование документации, чтобы объяснить, какой шаблон проектирования применялся

```
/**
* Mediator for validating Form objects using a generic
* Validator object
*/
class FormValidation
{
// ...
}
```

Пакет-посредник также известен как пакет-*мост*. Мост соединяет два пакета, которые станут более полезными при совместном использовании, но они не должны знать о существовании или внутренней работе друг друга.

В некоторых случаях даже имеет смысл назвать такой пакет пакетом-*адаптером* в том случае, если бы посредник был определен внутри пакета форм как интерфейс, например `FormValidationMediatorInterface`. Затем можно было бы создать пакет-адаптер, содержащий реализацию интерфейса `FormValidationMediatorInterface`, используя конкретную библиотеку для валидации, которую применяли в этом примере. Введение данного интерфейса также позволило бы пользователям пакета форм применять собственный любимый валидатор, написав единственный класс-адаптер. Возможно, это напомнило вам предыдущий пример, где используется интерфейс `HandlerInterface` из пакета `monolog`, для которого пользователи также могут предоставить собственную реализацию адаптера и, при желании, распространять ее как пакет.

Цепочка обязанностей

Еще один полезный шаблон, который может помочь нам в нашем стремлении разорвать цикл зависимостей, – это *Цепочка обязанностей*[1]. Его можно использовать, чтобы позволить другим частям приложения подключаться к определенному процессу и позволить им делать все, что они хотят (*см.* листинг 9-17).

Листинг 9-17. Использование шаблона Цепочка обязанностей

```
interface FormValidatorInterface
{
public function validate(FormInterface $form);
}
class Form implements FormInterface
{
private $validators = [];
public function addValidator(
FormValidatorInterface $validator
): void {
$this->validators[] = $validator;
}
public function isValid(): bool
{
foreach ($this->validators as $validator) {
$validator->validate($this);
}
// ...
}
}
```

Любой класс, реализующий `FormValidatorInterface`, можно добавить в стек валидаторов. Это прекрасный пример применения принципа *открытости/закрытости* (глава 2) к классу `Form`. Можно изменить поведение класса относительно валидации, просто внедрив в него другие объекты вместо изменения его кода.

Обратите внимание, что оригинальный рецепт *Цепочки обязанностей* включает в себя отдельный объект для запроса, и каждый из кандидатов для этого запроса должен явно передавать объект запроса следующему кандидату. В большинстве случаев такая реализация слишком сложна. Простой перебор кандидатов имеет гораздо больше смысла и определенно более удобен для чтения.

---

[1] *Erich Gamma e. a.* Design Patterns: Elements of Reusable Object-Oriented Software. Addison-Wesley, 1994.

Использование цепочки объектов с одинаковой ответственностью является отличным способом уменьшить зацепление пакетов. Пакет может содержать только один интерфейс, от которого могут зависеть другие пакеты (т. е. реализовывать). Проект, который использует эти пакеты, должен лишь правильно настроить граф объекта. См. рис. 9-13.

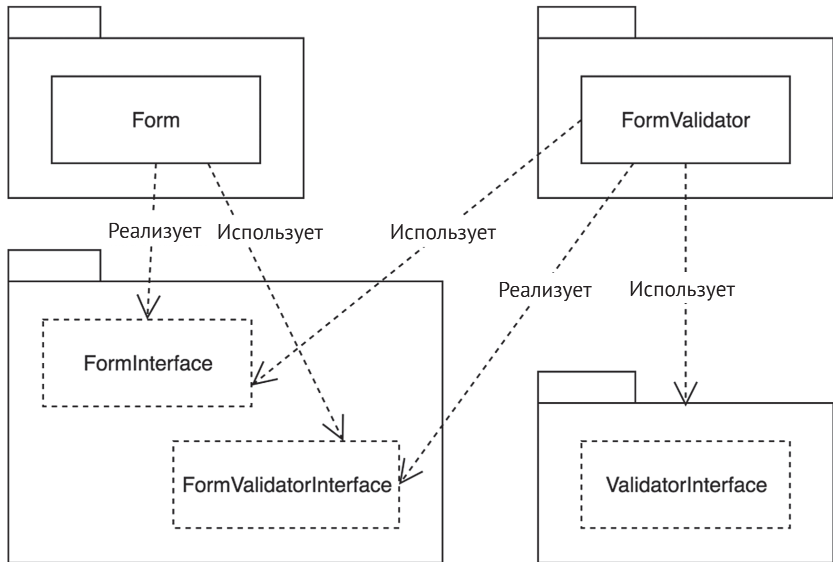


Рис. 9-13. Цепочка обязанностей используется для разрыва цикла зависимостей

Сочетание Посредника и Цепочки обязанностей: система событий

Существует одно из последних решений, которое является очень распространенным и подходящим способом устранения циклов зависимости. Это комбинация *Посредника* и *Цепочки обязанностей*, которую часто называют *диспетчером*, или *менеджером*, *событий*.

Диспетчер событий – это абстрактный посредник: имена и типы сообщений не определяются заранее. Он просто передает сообщения (события) делегатам (слушателям событий). В листинге 9-18 вы найдете простую реализацию такого диспетчера событий, которая позволяет регистрировать слушателей событий (которые должны реализовывать `ListenerInterface`) с использованием имени события, которое они хотят прослушивать.

Листинг 9-18. Реализация класса EventDispatcher и интерфейса ListenerInterface

```
class EventDispatcher
{
private $listeners = [];
public function registerListener(
string $eventName,
ListenerInterface $listener
): void {
$this->listeners[$eventName][] = $listener;
}
public function dispatch(
string $eventName,
$eventData
): void {
foreach ($this->getListeners($eventName) as $listener) {
$listener->notify($eventData);
}
}
private function getListeners($eventName): array
{
if (isset($this->listeners[$eventName])) {
return $this->listeners[$eventName];
}
// Слушатели этого события не определены;
return [];
}
}
interface ListenerInterface
{
public function notify($eventData): void;
}
```

В классе `Form` можно использовать диспетчер событий, чтобы инициировать событие всякий раз после отправки формы. Назовем это событие `form.submitted` (*см.* листинг 9-19).

Листинг 9-19. Класс Form инициирует событие при отправке

```
class Form
{
private $eventDispatcher;
public function __construct(EventDispatcher $eventDispatcher)
{
$this->eventDispatcher = $eventDispatcher;
}
public function submit(array $data): void
```

```
{
// Создаем объект события, предоставляем правильный контекст;
$event = new FormSubmittedEvent($this, $data);
$this->eventDispatcher
->dispatch('form.submitted', $event);
}
// ...
}
```

Класс `FormSubmittedEvent` довольно прост. Он используется только для переноса некоторых контекстных данных о произошедшем событии. В этом случае он позволяет слушателям событий проверять (и изменять) сам объект формы и данные, которые были отправлены (*см.* листинг 9-20).

Листинг 9-20. Класс события, которое инициируется при отправке формы

```
class FormSubmittedEvent
{
private $form;
public function __construct(FormInterface $form, array $data)
{
$this->form = $form;
}
public function getForm(): FormInterface
{
return $this->form;
}
public function getData(): array
{
return $this->data;
}
}
```

Теперь нам нужно только реализовать слушателя событий, который проверяет форму на основе представленных данных. Он слушает событие `form.submitted` и распаковывает объект `FormSubmittedEvent`. Если какие-то из представленных данных из объекта события не являются валидными, слушатель добавляет ошибку в объект формы (*см.* листинг 9-21).

Листинг 9-21. Слушатель событий, который проверяет данные формы

```
class ValidateDataOnFormSubmitListener
{
```

```
public function __construct(ValidatorInterface $validator)
{
$this->validator = $validator;
}
public function notify(FormSubmittedEvent $event): void
{
$form = $event->getForm();
$submittedData = $event->getData();
if (!$this->validator->validate(...)) {
//Часть представленных данных недействительна;
$form->addError(...);
}
}
}
```

Чтобы все это работало, нужно настроить диспетчер событий и зарегистрировать слушателя валидации формы, а затем предоставить диспетчера событий в качестве аргумента конструктора объекта `Form` (*см.* листинг 9-22).

Листинг 9-22. Настройка диспетчера событий, слушателя и формы

```
$eventDispatcher = new EventDispatcher();
$validationListener = new ValidateDataOnFormSubmitListener();
$eventDispatcher->registerListener(
'form.submitted',
$validationListener
);
$form = new Form($eventDispatcher);
//Инициирует слушателя событий;
$form->submit([...]);
```

`EventDispatcher` является надлежащим *посредником*: `Form` никогда не общается с `Validator` напрямую, а всегда использует для этой цели `EventDispatcher`. Внутри `EventDispatcher` слушатели образуют *Цепочку обязанностей*. Каждый из них получает шанс ответить на событие `form.submitted`.

Глядя на диаграмму зависимостей (*см.* рис. 9-14), видим, что кругов нет.

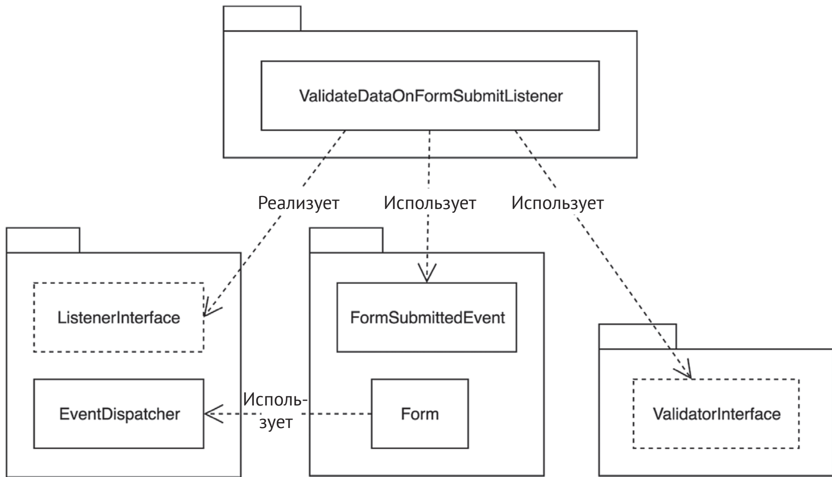


Рис. 9-14. Новая диаграмма зависимостей, включающая диспетчер событий

Возможно, на этой диаграмме отсутствует связь между диспетчером событий и слушателем событий. В некотором смысле EventDispatcher должен зависеть от класса ValidateDataOnFormSubmitListener. Однако это лишь зависимость времени выполнения. Если мы посмотрим на код, то *строгой* зависимости от класса ValidateDataOnFormSubmitListener нет. Можно использовать класс EventDispatcher без этого конкретного слушателя событий. Следовательно, это не должно приводить к зависимости между пакетами.

Использование событий для уменьшения зацепления пакетов может быть очень эффективным. Однако тут есть и недостатки. Во-первых, диспетчер событий является *высокоабстрактным посредником*. В вызове EventDispatcher::dispatch() ничего не говорится о том, что должна произойти валидация. В то же время валидация представляет собой концепцию, которая играет ключевую роль в обработке форм, поэтому работа с ней «за кулисами» в слушателе событий – возможно, не самое лучшее решение. Более подходящим решением в этом случае было бы применение инверсии зависимостей. Используйте интерфейс FormValidatorInterface и пакет-адаптер с реализацией этого интерфейса, который знает, как использовать класс Validator, в результате чего циклическая зависимость будет эффективно удалена.

Существуют и другие ситуации, когда использование подсистемы событий было бы разумным решением. Например, когда вы

хотите разрешить пользователям получать уведомления, когда процесс входит или выходит из определенной фазы. В этом случае всегда стремитесь смоделировать сами события так, чтобы они были неизменными. Это должно предотвратить трудности с отладкой, связанные с состоянием, которые неожиданным образом изменяются внутри слушателей событий.

Если вы действительно хотите разрешить слушателям событий изменять данные событий, посмотрите, нельзя ли вместо этого использовать нечто вроде механизма фильтра или конвейера. Это позволяет избегать абстрактности имени «диспетчер событий» и вводить некоторые более значимые имена классов/интерфейсов, такие как «фильтр данных формы», «препроцессор запроса», «сборщик заголовка ответа» и т. д.

Заключение

В этой главе мы обсудили множество аспектов зацепления. Сначала мы рассмотрели различные типы зависимостей между классами, которые могут привести к пакетным зависимостям, когда зависимость одного класса от другого выходит за границы пакета, содержащего класс.

Следуя по пути от зависимости к зависимости, вы иногда возвращаетесь к пакету, с которого начали. В этом случае у вас есть цикл в вашем графе зависимостей. Принцип *ацикличности зависимостей* говорит нам, что в нашем графе зависимостей не должно быть циклов. Размышляя об этом, мы пришли к выводу, что циклы действительно вызывают множество проблем, и к концу этой главы мы узнали, что их не так сложно преодолеть.

Разрыв всех циклов в нашем графе зависимостей (который в таком случае является *ациклическим ориентированным графом*) позволяет нам легко создавать ветви в истории пакетов. Это означает, что мы можем работать над новыми и даже старшими версиями пакета, не препятствуя прогрессу других пакетов или вообще выпуская их следующую старшую версию. Отсутствие циклов означает, что изменение в одном из пакетов влияет только на наименьшее количество возможных пакетов.

Подверженность изменениям в других пакетах или необходимость изменения других пакетов при изменении вашего пакета является предметом последующих двух глав.

Глава 10

Принцип устойчивых зависимостей

В предыдущей главе мы обсудили принцип *ацикличности зависимостей*, который помогает нам предотвращать образование циклов в наших графах зависимостей. Самая большая опасность циклических зависимостей заключается в том, что проблемы в одной из ваших зависимостей могут иметь неприятные последствия, после того как они пройдут весь цикл через граф зависимостей.

Даже если в вашем графе зависимостей нет циклов, существует вероятность того, что зависимости пакета начнут создавать проблемы в будущем. Всякий раз, когда вы обновляете одну из зависимостей вашего проекта, вы надеетесь, что ваш проект будет работать так же, как и прежде. Однако всегда существует риск того, что он неожиданно начнет давать сбои.

Когда ваш проект по-прежнему работает после обновления его зависимостей, разработчики этих зависимостей, вероятно, знают, что многие пакеты *зависят от их пакета*.

Поэтому в каждом патч-релизе или младшем выпуске они будут только исправлять ошибки или добавлять новые функции. Они никогда не отправляют изменения, которые могут вызвать сбой в зависимом пакете.

Однако если что-то в вашем проекте внезапно сломалось после обновления одной из зависимостей, ответственные за поддержку пакета, очевидно, внесли некоторые изменения, которые не являются обратно совместимыми. Такие изменения всплывают на графике зависимостей и приводят к проблемам в зависимых пакетах.

Когда зависимость вашего проекта внезапно вызывает сбои, для начала нужно переосмыслить свой выбор зависимостей, вместо того чтобы обвинять сопровождающих. Некоторые пакеты очень изменчивы, а некоторые нет. По своей природе пакет может часто меняться по любой причине. Возможно, эти изменения связаны с проблемной областью или с одной из ее зависимостей.

Аналогично, прежде чем добавлять зависимость в свой проект, необходимо решить: есть вероятность, что эта зависимость изменится? *Легко* ли его сопровождающим изменить ее? Другими словами, можно ли считать зависимость *устойчивой* или она не является таковой?

СЕМАНТИЧЕСКОЕ ВЕРСИОНИРОВАНИЕ И УСТОЙЧИВОСТЬ

Как обсуждалось в главе, посвященной принципу эквивалентности освобождения / повторного использования (глава 6), слово «устойчивый» также используется в контексте семантического контроля версий. Пакет считается устойчивым, если имеет версию не ниже 1.0.0 и не находится в ветке разработки (или стадиях альфа, бета, выпуск-кандидат). Такая устойчивая версия обещает иметь общедоступный API, который не изменяется обратно несовместимыми способами.

Принцип *устойчивых зависимостей* также касается устойчивости пакета, но не обязательно связан с семантическим версионированием. В этой главе «устойчивый» означает «вряд ли изменится». Устойчивый пакет в этом контексте – это пакет, от которого зависит множество других пакетов, хотя сам он не зависит от других пакетов.

Устойчивость

Устойчивость пакета зависит от того, насколько легко что-то изменить в его коде. Речь идет не о чистом коде или о том, можно ли легко изменить код. Речь идет о том, насколько пакет *ответствен* по отношению к другим пакетам и подвержен ли он изменениям в какой-либо из своих зависимостей.

Изменения в зависимостях пакета могут быть связаны с самим пакетом. Вам часто нужно будет вносить изменения в собственный пакет, чтобы учесть изменения в его зависимостях. Если у вас

много зависимостей, более вероятно, что обновление ваших зависимостей потребует от вас изменения вашего пакета. Такой пакет будет называться *зависимым пакетом* (*см.* рис. 10-1). Когда пакет необходимо часто менять, чтобы учесть изменение одной из его зависимостей, его следует рассматривать как *неустойчивый* пакет.

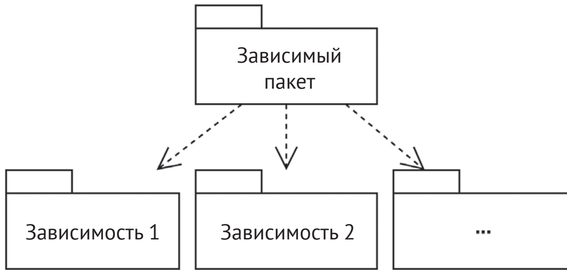


Рис. 10-1. Сильно зависимый пакет

Если у пакета нет зависимостей или их немного, есть вероятность, что обновление ваших зависимостей не вызовет никаких проблем. Данный пакет называется *независимым* (*см.* рис. 10-2). Такой пакет не очень подвержен изменениям в его зависимостях, поэтому его следует считать *устойчивым*.

Рис. 10-2. Независимый пакет

Есть еще одно направление в графе зависимостей, которое необходимо учитывать: направление *к* пакету. Другими словами, сколько других пакетов зависит от данного пакета? Если их число велико, будет сложно внести изменения в пакет, потому что от него зависит столько пакетов, и эти локальные изменения могут потребовать множества модификаций *где-то еще*. Такой пакет называется *ответственным* (*см.* рис. 10-3).

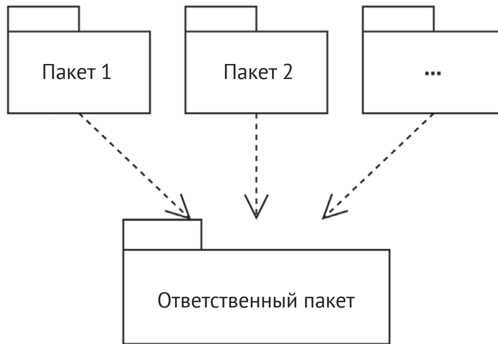


Рис. 10-3. Ответственный пакет – это пакет, от которого зависит много пакетов

С другой стороны, если количество входящих зависимостей невелико или даже отсутствует, ответственному за поддержку пакета будет очень легко внести в него изменения, поскольку эти изменения будут иметь незначительное влияние на другие пакеты. Мы называем пакет, от которого не зависят другие пакеты, *неответственным*, потому что он не будет нести ответственность за какие-либо проделанные с ним изменения (*см.* рис. 10-4).

Рис. 10-4. Неответственный пакет, у которого нет зависящих от него пакетов

Человек, отвечающий за поддержку такого пакета, волен изменять все, что ему заблагорассудится. Напротив, пакет с большим количеством зависящих от него пакетов можно назвать *ответственным*, поскольку отвечающий за его поддержку не может просто изменить то, что он хочет. Ожидается, что любое изменение окажет влияние на зависимые пакеты.

На этом этапе имеет смысл не только учитывать количество пакетов, зависящих от вашего пакета, но также учитывать и количество приложений, которые зависят от него. Будучи разработчиком пакетов, вы не всегда можете получить точное представление об этом, но менеджеры пакетов обычно отслеживают количество скачиваний пакета.

Если оно высокое, можете быть уверены, что у пакета много пользователей. В этом случае у вас есть ответственный пакет, то есть он должен быть стабильным для своих пользователей.

Не каждый пакет может быть высокостабильным

Пакеты, которые являются более независимыми и ответственными, должны считаться *высокостабильными*. Это пакеты, которые не нужно менять из-за изменения одной из их зависимостей, но они также не могут легко измениться сами, потому что другие пакеты сильно зависят от них. См. рис. 10-5.

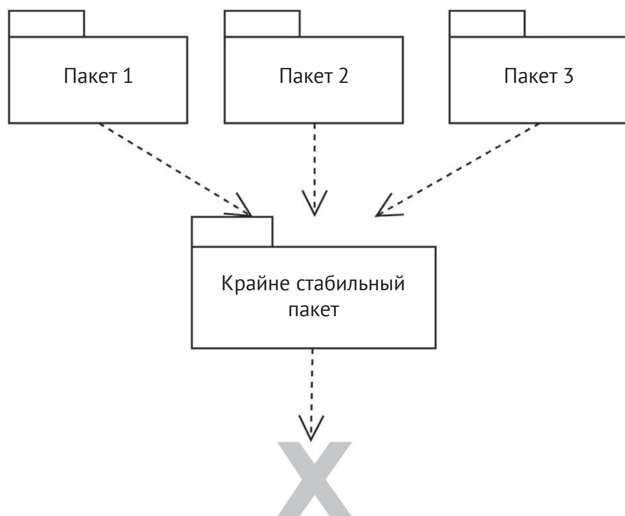


Рис. 10-5. Высокостабильный пакет: никаких зависимостей, только зависимые пакеты

Эти высокостабильные пакеты обычно представляют собой небольшие библиотеки кода, реализующие абстрактные концепции, которые полезны во многих различных контекстах.

С другой стороны, шкалы, пакеты, которые являются более зависимыми, но в то же время крайне неответственными, следует считать *крайне нестабильными*. Эти пакеты подвержены изменениям в любой из своих зависимостей, но они не зависят ни от

какого другого пакета, поэтому у них нет проблем с изменением, поскольку изменение не будет сквозным. См. рис. 10-6.

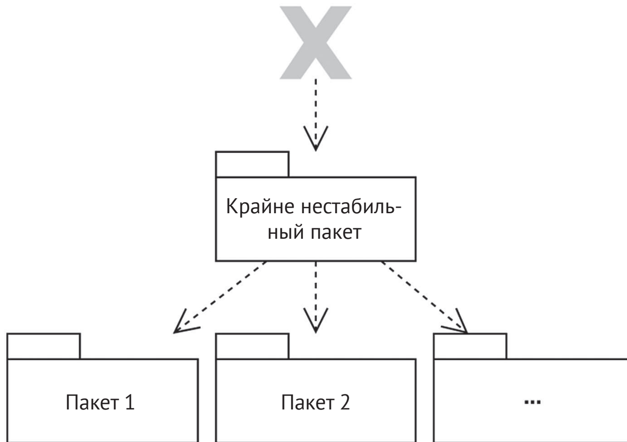


Рис. 10-6. Крайне нестабильный пакет: много зависимостей, зависимых пакетов нет

Эти крайне нестабильные пакеты, вероятно, содержат конкретные реализации, которые, например, связаны с конкретной библиотекой постоянства, или могут содержать подробные реализации бизнес-правил, которые могут измениться. Код, который полезен только в контексте определенного каркаса приложения, вероятно, также находится внутри нестабильного пакета, поскольку сам каркас является крайне нестабильным в соответствии с определением, использованным в этой главе.

Наконец, есть пакеты, у которых нет зависимостей, но другие пакеты (или приложения) также не зависят от них. Такие пакеты являются независимыми и неответственными. Это бесполезные пакеты. Большинство пакетов, однако, находятся где-то между *крайне независимыми и ответственными*, а также *очень зависимыми и неответственными*.

Нестабильные пакеты должны зависеть только от более стабильных

Интуитивно, если бы нестабильный пакет зависел от стабильного, это было бы не страшно. В конце концов, стабильный пакет вряд

ли окажет негативное влияние на пакет, который уже нестабилен. Однако, с другой стороны, *стабильный* пакет, который зависит от *нестабильного*, будет не приемлем. Нестабильный пакет создает угрозу устойчивости стабильного пакета и фактически делает его менее стабильным.

Чтобы запретить разработчикам пакетов использовать «плохие» зависимости, принцип *устойчивых зависимостей* гласит:

> *Зависимости между пакетами в проекте должны быть направлены в сторону устойчивости пакетов. Пакет должен зависеть только от тех пакетов, которые более устойчивы, чем он*[1].

Другими словами, менее стабильные пакеты могут зависеть от более стабильных. Стабильные пакеты не должны зависеть от нестабильных.

Оценка устойчивости

Устойчивость – это фактически измеряемая единица, которую мы можем использовать, чтобы определить, соответствует ли пакет в графе зависимостей принципу *устойчивых зависимостей*.

Обычный способ выражения устойчивости – это расчет метрики `I` для пакетов. Сначала нужно посчитать количество классов *вне* пакета, которые зависят от класса *внутри* пакета. Мы называем это значение `C-in`. Затем нужно посчитать количество классов *вне* пакета, от которых зависит класс *внутри* пакета. Мы называем это значение `C-out`.

Потом можно определить `I`-метрику пакета путем вычисления значения `C-out`, деленного на `C-in + C-out`. Это означает, что `I` будет между 0 и 1, где 1 указывает на то, что пакет максимально нестабилен, а 0 означает, что он максимально стабилен.

Высокостабильный пакет является *ответственным*: у него много зависящих от него пакетов, поэтому `C-in` – большое число. В то же время он *независим*: у него нет зависимостей, поэтому `C-out = 0`. Это означает, что `I = 0`, поскольку `C-out / (C-in + C-out) = 0`.

Крайне *нестабильный* пакет очень *зависим*: у него много зависимостей, поэтому `C-out` – большое число. Но он также является *неот-*

---

[1] *Robert C. Martin.* Engineering Notebook. C++ Report, Feb 1997 (PDF-версия доступна по адресу http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod).

ветственным: у него нет других пакетов, которые зависят от него, поэтому `C-in = 0`. Тогда `I = 1`, поскольку `C-out / (C-in + C-out) = 1`.

Конечно, это очень экстремальные примеры. У большинства пакетов есть значение I, которое не равно ни 0, ни 1, а находится где-то посередине. Например, у пакета в центре рис. 10-7 `C-out` равно 3, `C-in` – 2, поэтому значение `I` для этого пакета составляет `3 / (2 + 3) = 0,6`. Это означает, что пакет следует считать относительно нестабильным; количество исходящих зависимостей выше, чем количество входящих.

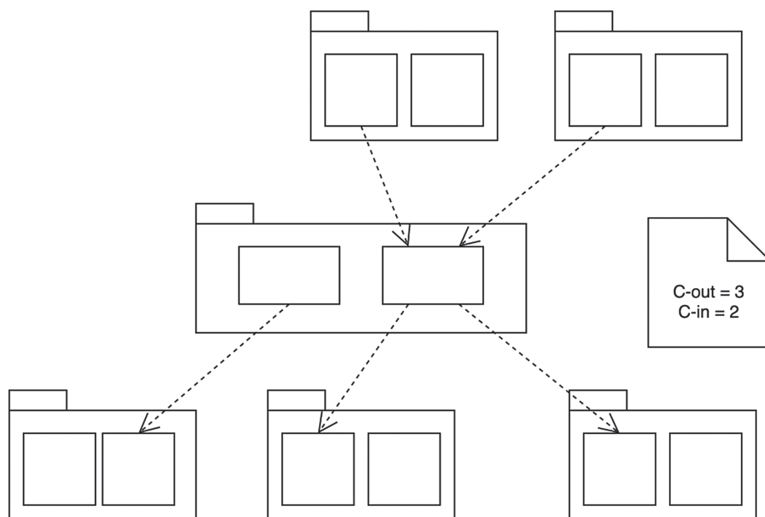


Рис. 10-7. Расчет значений C-in и C-out для пакета в центре

Снижение нестабильности и повышение устойчивости

Согласно принципу *устойчивых зависимостей*, зависимости между пакетами в проекте должны быть направлены «в сторону устойчивости пакетов». Другими словами, каждый наш шаг в графе зависимостей должен приводить к более стабильному пакету. Более стабильный также означает менее нестабильный, поэтому нам разрешено предпринимать в графе зависимостей только те шаги, которые приводят к пакетам с *более низким* значением I (*см.* рис. 10-8).

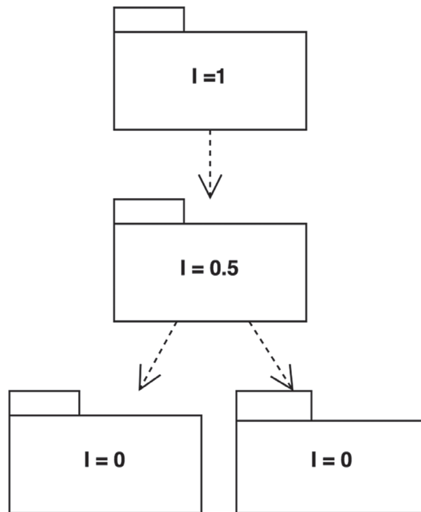


Рис. 10-8. Пример, где все зависимости направлены в сторону устойчивости

Когда вы рисуете такую диаграмму для своих пакетов, целесообразно помещать пакеты с низким значением I возле нижней части, а пакеты с высоким значением I возле верхней. Тогда каждая стрелка зависимости должна указывать вниз, поскольку это обозначает направление устойчивости. Если стрелка будет указывать вверх, как на рис. 10-9, значит, принцип *устойчивых зависимостей* был нарушен (позже мы обсудим варианты, как заставить стрелку снова показывать вправо).

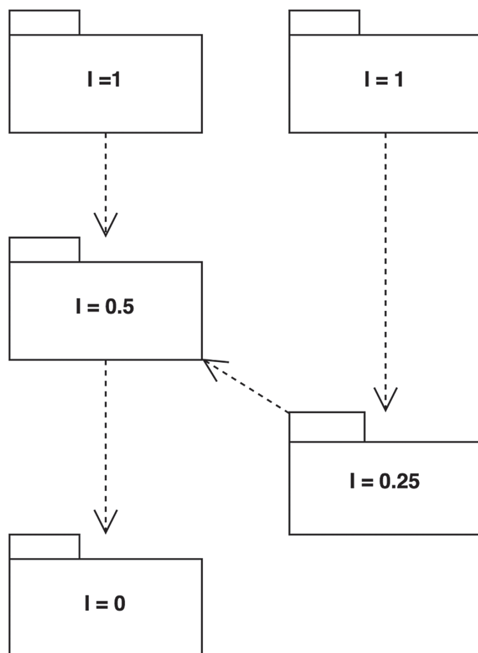


Рис. 10-9. Пример, где не все зависимости направлены в сторону устойчивости

Вопрос: следует ли учитывать все пакеты, какие есть во вселенной?

Это интересный вопрос. Чем больше пакетов зависит от пакета, тем более ответственным он будет и, следовательно, более стабильным он станет. Но при расчете метрик зацепления пакетов было бы практически невозможно принять во внимание все другие пакеты и приложения, которые существуют в мире. Таким образом, когда мы вычисляем метрику I, а затем метрику A, мы можем и должны смотреть только на все те пакеты, что установлены в данном приложении. Мы можем поместить их все в одну большую диаграмму зависимостей и начать проверять, насколько хорошо они следуют принципам зацепления.

В следующих разделах мы обсудим нарушения принципа *устойчивых зависимостей* и способы их устранения (если у вас есть возможность для этого!).

Нарушение: ваш стабильный пакет зависит от стороннего нестабильного пакета

В следующем примере я использую библиотеку Gaufrette (https://github.com/KnpLabs/Gaufrette), которая предлагает уровень абстракции для файловых систем. Она позволяет переключаться с локальной файловой системы на файловую систему в памяти или даже на хранилище Dropbox или Amazon, без необходимости вносить изменения в свой собственный код.

Класс `FileCopy` в листинге 10-1 является частью моего собственного пакета. Эта примитивная реализация механизма копирования позволяет копировать файлы между любыми двумя файловыми системами. Она зависит от класса `Filesystem`, предлагаемого библиотекой Gaufrette.

Листинг 10-1. Класс FileCopy

```
use Gaufrette\Filesystem as GaufretteFilesystem
class FileCopy
{
private $source;
private $target;
public function __construct(
GaufretteFilesystem $source,
GaufretteFilesystem $target
) {
$this->source = $source;
$this->target = $target;
}
public function copy($filename)
{
$fileContents = $this->source->get($filename);
$this->target->write($filename, $fileContents);
}
}
```

Пакет, содержащий класс `FileCopy`, назовем его `filesystem-manipulation`, имеет явную зависимость от пакета `knplabs/gaufrette`, который содержит класс `Filesystem`, как показано в листинге 10-2.

Листинг 10-2. Список зависимостей пакета filesystem-manipulation

```
{
"name": "filesystem-manipulation",
"require": {
```

```
"knplabs/gaufrette": "~0.1"
}
}
```

В настоящее время FileCopy является единственным классом в этом пакете. У него есть одна зависимость от класса из другого пакета, а это приводит к тому, что значение C-out этого пакета равно 1. В проекте, где используется пакет filesystem-manipulation, есть один класс, который использует класс FileCopy, поэтому значение C-in также равно 1, следовательно, I = 1 / (1 + 1) = 0,5.

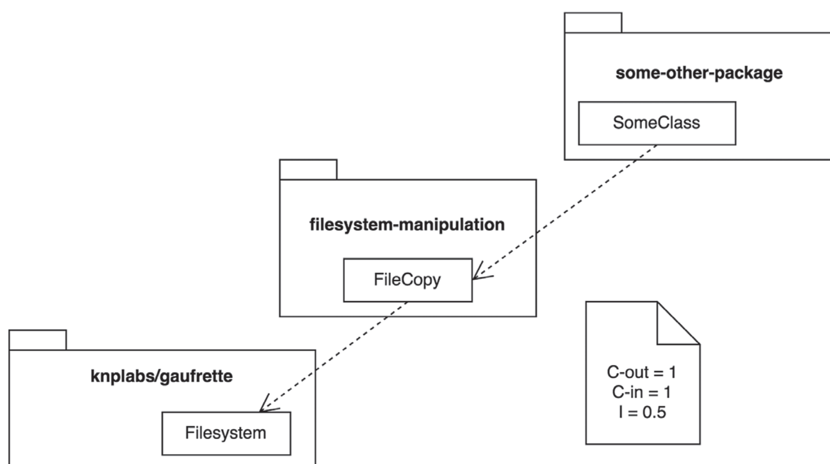


Рис. 10-10. Вычисление значения I для filesystem-manipulation

Когда мы проводим то же самое вычисление для пакета knplabs/gaufrette, нам нужно подсчитать количество классов за пределами этого пакета, которые зависят от классов внутри пакета. Этот пакет содержит множество классов-адаптеров, которые заставят абстракцию файловой системы работать со всеми типами решений для внешнего хранения данных. Все эти классы нуждаются в дополнительных зависимостях для выполнения своей работы. Этим объясняется большое количество исходящих зависимостей, которое после подсчета оказывается равным 54. Таким образом, C-out = 54. В рамках текущего проекта только класс FileCopy зависит от одного из классов knplabs/gaufrette, поэтому C-in = 1. Это приводит к довольно высокому значению I, а именно 54 / (54 + 1) = 54/55, что почти равно 1.

Таким образом, knplabs/gaufrette оказывается *крайне нестабильным* пакетом. Гораздо более нестабильным, чем наш собственный пакет filesystem-manipulation. Тем не менее пакет filesystem-manipulation зависит от всего пакета knplabs/gaufrette.

Поэтому мы явно нарушаем принцип *устойчивых зависимостей*, поскольку не все наши зависимости направлены в сторону устойчивости. Наоборот, мы наблюдаем, что они направлены в сторону *неустойчивости*. Это становится еще более явным, когда мы упорядочиваем пакеты в соответствии с их стабильностью, а затем рисуем стрелки зависимости (*см.* рис. 10-11).

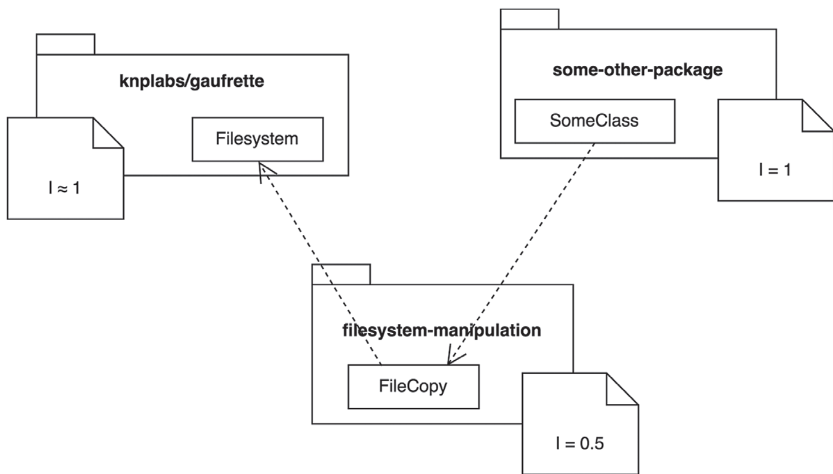


Рис. 10-11. Пакет filesystem-manipulation зависит от менее стабильного пакета

Причина, по которой knplabs/gaufrette является таким нестабильным пакетом, заключается в том, что он содержит множество конкретных адаптеров файловой системы для Dropbox, Amazon S3, SFTP и т. д. Эти адаптеры не используются всеми одновременно. Таким образом, согласно принципу *совместного повторного использования*, они должны находиться в отдельных пакетах.

Пакету filesystem-manipulation не нужны все эти специфические адаптеры файловой системы; ему нужен только класс Filesystem, который предоставляет общие методы для связи с любой специфической файловой системой.

Решение: используйте инверсию зависимостей

Чтобы исправить граф зависимостей и изменить направление стрелок, дабы они указывали в направлении устойчивости, нам бы очень хотелось взять класс `Filesystem` (который является *фактическим* уровнем абстракции файловой системы) и поместить его в отдельный пакет: `knplabs/gaufrette-filesystem-abstraction`. Тогда классы-адаптеры должны быть размещены внутри других пакетов, таких как `knplabs/gaufrette-amazon-adapter`, `knplabs-/gaufrette-sftp-adapter` и т. д. Затем можно было бы изменить зависимость от `knplabs/gaufrette` на `knplabs/gaufrette-filesystem-abstraction`, и это было бы как раз то, что нужно.

Тем не менее мы не можем этого сделать, поскольку не отвечаем за поддержку `knplabs/gaufrette`. Поэтому нам нужно прибегнуть к другому решению, о котором мы уже говорили: нам следует применить принцип *инверсии зависимостей*. Во-первых, вместо зависимости от класса `Gaufrette\Filesystem`, который по-прежнему находится в крайне нестабильном пакете, мы определяем свой собственный интерфейсе `FilesystemInterface` (*см*. листинг 10-3) внутри нашего пакета `filesystem-manipulation`.

Листинг 10-3. Интерфейс FilesystemInterface

```
interface FilesystemInterface
{
public function read($path): string;
public function write($path, $contents): void;
}
```

Затем мы позволяем конструктору `FileCopy` принимать объекты, которые реализуют этот новый интерфейс (*см*. листинг 10-4).

Листинг 10-4. Класс FileCopy использует новый интерфейс FilesystemInterface

```
class FileCopy
{
// ...
public function __construct(
FilesystemInterface $source,
FilesystemInterface $target
) {
// ...
}
// ...
}
```

Теперь мы можем фактически удалить зависимость от `knplabs/gaufrette` из определения нашего пакета `filesystem-manipulation`. По сути, пакет сразу стал независимым: у него вообще нет зависимостей. Это означает, что теперь значение I равно 0, и его следует считать очень стабильным.

Как уже упоминалось, мы по-прежнему хотим использовать библиотеку Gaufrette. Поэтому нам нужно преодолеть разрыв между `FilesystemInterface` и классом `Gaufrette\Filesystem`. Это можно сделать, введя новый класс под названием `GaufretteFilesystemAdapter` (*см.* листинг 10-5).

Листинг 10-5. GaufretteFilesystemAdapter

```
use Gaufrette\Filesystem as GaufretteFilesystem;
class GaufretteFilesystemAdapter implements FilesystemInterface
{
private $gaufretteFilesystem;
public function __construct(
GaufretteFilesystem $gaufretteFilesystem
) {
$this->gaufretteFilesystem = $gaufretteFilesystem;
}
public function read($path): string
{
return $this->gaufretteFilesystem->get($path);
}
public function write($path, $contents): void
{
$this->gaufretteFilesystem->write($path, $contents);
}
}
```

Этот класс использует объект файловой системы Gaufrette по композиции и в то же время является подходящей заменой для `FilesystemInterface`. Мы поместили этот класс в новый пакет, `gaufrettefilesystem-adapter`. Поскольку классу нужны и интерфейс `FilesystemInterface`, и класс `Gaufrette\Filesystem`, он зависит как от пакета `knplabs/gaufrette`, так и от пакета `filesystem-manipulation` (*см.* листинг 10-6).

Листинг 10-6. Список зависимостей пакета gaufrette-filesystem-adapter

```
{
"name": "gaufrette-filesystem-adapter",
"require": {
"knplabs/gaufrette": "1.*"
```

```
"filesystem-manipulation": "*"
}
}
```

Значение `C-out` этого нового пакета `gaufrette-filesystem-adapter` равно 2, потому что он использует два класса за пределами пакета. Его значение `C-in` равно 0, поскольку ни один другой пакет не использует класс из этого пакета. Это означает, что `I = 2 / (2 + 0) = 1`. Он крайне нестабилен (то есть его легко изменить), что вполне нормально для пакета адаптера.

Посмотрите на рис. 10-12, чтобы узнать, что все это повлияло на граф зависимостей и стрелки в нем.

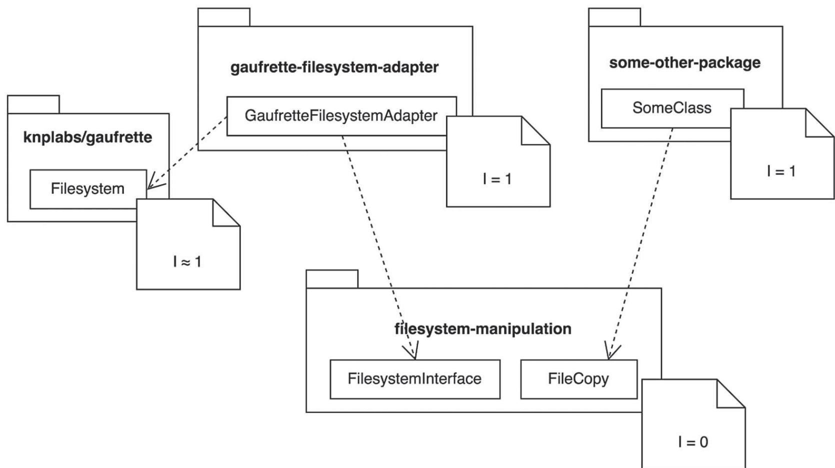


Рис. 10-12. Каждая зависимость направлена в сторону устойчивости

Пакеты теперь рассортированы в направлении устойчивости, и все стрелки зависимостей указывают вниз, а это означает, что ни один пакет в этой системе больше не нарушает принцип *устойчивых зависимостей*.

Все это было сделано без внесения каких-либо изменений в сторонний код. Мы применили принцип *инверсии зависимостей* к классу `FileCopy`, позволив ему зависеть от чего-то абстрактного, а не от чего-то конкретного, что автоматически делает класс `FileCopy` легко расширяемым: другие могут реализовать собственные адаптеры для `Filesystem` и сделать его совместимым, например,

с библиотекой абстракций файловой системы `Filesystem`[1]. Это также делает пакет `filesystem-manipulation` более легким в обслуживании, поскольку изменения в `knplabs/gaufrette` больше не влияют на него.

Отсутствие влияния внешних изменений делает пакет `filesystem-manipulation` очень стабильным. Он вряд ли изменится из-за своих зависимостей (поскольку у него их больше нет). Его прежняя нестабильность отталкивается к более конкретному пакету `gaufrette-filesystem-adapter`, который теперь подвержен изменениям в `knplabs/gaufrette`. Но даже несмотря на то, что код внутри пакета `gaufrette-filesystem-adapter` может измениться, он не представляет угрозы для других частей системы, поскольку от него не зависит ни один другой пакет.

Пакет может быть как ответственным, так и наоборот

Как я уже вскользь упоминал, у пакета `knplabs/gaufrette` имеются проблемы с дизайном. Он содержит классы, которые не будут использоваться теми, кто использует пакет в своем проекте, поэтому он нарушает принцип *совместного повторного использования*. Он также содержит классы (фактически одни и те же классы), которые не закрыты относительно изменений одного и того же типа, поэтому пакет нарушает принцип *общей* закрытости.

Теперь, когда мы смотрим на пакет `knplabs/gaufrette` с точки зрения устойчивости, становится ясно, что группировка тех классов, которые на самом деле не связаны друг с другом, является причиной, по которой этот пакет стал очень нестабильным. Он вводит много внешних зависимостей, поэтому другие пакеты больше не могут зависеть от него: это небезопасно.

Это плохое свойство для пакетов, которые должны использоваться многократно. На самом деле надежный пакет должен быть очень безопасным, чтобы от него можно было зависеть: он должен быть стабилен. Другими словами, он должен быть независимым и ответственным.

В предыдущем разделе мы обсудили решение этой проблемы устойчивости. Это повлекло за собой введение интерфейса и адаптера для изменения направления зависимостей. Нам нужно было

[1] https://github.com/thephpleague/flysystem.

прибегнуть к этому решению, потому что мы не могли сделать то, что действительно было необходимо, – разбить пакет на пакет, содержащий более широко повторно используемые части (например, класс `Gaufrette\Filesystem` и интерфейс `Gaufrette\Adapter`) и один или несколько пакетов, содержащих более специфичные и конкретные части (например, адаптеры файловой системы для SFTP, Dropbox и т. д.).

Первый пакет не будет иметь зависимостей, только зависимые от него пакеты, что сделает его независимым и ответственным, то есть очень стабильным. Можно назвать его `knplabs/gaufrette-filesystem-abstraction`. Другие пакеты будут названы в честь конкретных файловых систем, для которых они предоставили реализацию, например `knplabs/gaufrette-sftp-adapter`. Каждый из этих пакетов может иметь столько зависимостей, сколько необходимо для специфичной реализации файловой системы. И конечно, каждый из них будет зависеть от `knplabs/gaufrette-filesystem-abstraction`, потому что этот пакет будет содержать интерфейс, который должен реализовать каждый адаптер файловой системы. Это сделало бы эти пакеты адаптеров зависимыми и несколько менее неответственными. То есть количество зависящих от него пакетов будет меньше количества пакетов и приложений, зависящих от пакета абстракции основной файловой системы.

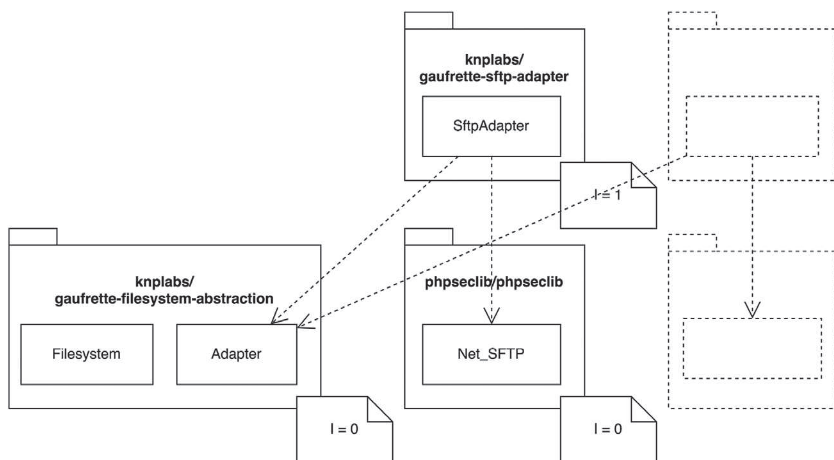


Рис. 10-13. Пакеты knplabs/gaufrette- * после рефакторинга

Самое замечательное состоит в том, что в таком созвездии пакетов `knplabs/gaufrette-filesystem-abstraction` будет очень стабильным, а пакет `filesystem-manipulation`, содержащий класс `FileCopy`, может безопасно зависеть от него. Сам по себе пакет `filesystem-manipulation` имеет значение I, равное 0,5, в то время как `knplabs/gaufrette-filesystem-abstraction` имеет значение I, равное 0, которое ниже. Все зависимости пакета будут следовать в направлении устойчивости, поэтому принцип *устойчивых зависимостей* не будет нарушен.

Заключение

Согласно принципу *устойчивых зависимостей*, зависимости должны быть направлены в сторону устойчивости. Это означает, что каждый пакет должен зависеть только от пакетов, которые более стабильны, чем он сам. Устойчивость пакета – это показатель того, как он вероятнее всего изменится.

Стабильный (устойчивый) пакет будет независимым (у него имеется лишь несколько зависимостей или вовсе ни одной) и ответственным (от него зависит множество классов). *Нестабильный* пакет будет зависимым (у него много зависимостей) и неответственным (от него не зависит ни один класс или зависит лишь несколько).

С помощью I-метрики (не)устойчивость можно определить количественно как `C-out/(C-out + C-in)`, где `C-out` – это число классов, от которых зависит пакет, а `C-in` – это число классов, которые зависят от класса в этом пакете. Если I ближе к 1, пакет будет относительно нестабильным. Если оно приближается к 0, он относительно стабилен.

Глава 11

Принцип устойчивых абстракций

Мы добрались до последнего из принципов проектирования, связанных с зацеплением пакетов, а это означает, что мы фактически достигли последнего из *всех* принципов разработки пакетов. Этот принцип, *принцип устойчивых абстракций*, касается устойчивости, равно как и принцип *устойчивых зависимостей*. В то время как предыдущий принцип гласит, что зависимости должны быть направлены «в сторону устойчивости», этот принцип гласит, что зависимости должны быть направлены в сторону *абстрактности*.

Устойчивость и абстрактность

Название принципа *устойчивых абстракций* содержит два важных слова: «устойчивый» и «абстрактный». Мы уже обсуждали устойчивость пакетов в предыдущей главе. Стабильный (устойчивый) пакет вряд ли сильно изменится. У него нет зависимостей, поэтому нет никаких внешних причин для его изменения. В то же время другие пакеты зависят от него. Поэтому пакет не должен меняться, чтобы не было проблем с этими зависимыми пакетами.

Из предыдущей главы мы узнали, что устойчивость можно рассчитать и убедиться, что граф зависимостей проекта содержит только зависимости растущей устойчивости или падающей неустойчивости. В этой главе мы узнаем, что мы также должны вычислить абстрактность пакетов и что направление зависимости должно быть одним из увеличения абстрактности или уменьшения конкретности.

Понятие абстрактности – это то, с чем мы также сталкивались в предыдущих главах. Например, из главы, посвященной принци-

пу *инверсии зависимостей* (глава 5), мы узнали, что наши классы должны зависеть от абстракций, а не от конкретных реализаций.

Мы обсудили несколько способов, с помощью которых класс может быть абстрактным. Наиболее очевидный способ – когда класс имеет абстрактные (также известные как виртуальные) методы. Эти методы должны быть определены в подклассе. Этот подкласс является конкретным классом, потому что представляет собой полную реализацию того типа вещей, который абстрактный класс пытается моделировать. Когда у класса есть только абстрактные методы, мы обычно называем его не классом, а интерфейсом. Классы, которые реализуют интерфейс, в конечном итоге должны обеспечить реализацию всех абстрактных методов, определенных в интерфейсе.

Принцип *инверсии зависимостей* говорит нам, что мы должны зависеть от абстракций, а не от конкретных реализаций. Причина, как и всегда, состояла в том, что мы должны быть готовы к изменениям. Класс, который зависит от чего-то конкретного, может меняться всякий раз, когда изменяется какая-то деталь реализации этой конкретной вещи (*см.* рис. 11-1). Кроме того, если в какой-то момент мы хотим заменить эту конкретную вещь на другую конкретную вещь, нам нужно будет изменить класс, чтобы понять и использовать эту новую конкретную вещь. И в данной ситуации это, вероятно, не единственный класс, который необходимо изменить.

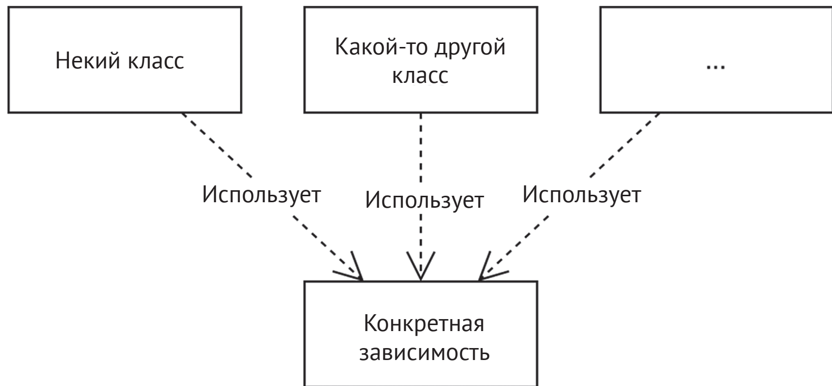


Рис. 11-1. Зависимость от конкретных вещей

Если вместо этого мы определяем нечто абстрактное, например абстрактный класс или предпочтительно интерфейс, и за-

висим от него, мы гораздо лучше подготовлены к изменениям (см. рис. 11-2). Большинство изменений происходят в конкретных вещах, то есть в полностью реализованных классах. Такие абстрактные вещи, как интерфейсы, будут оставаться такими же в течение более длительного периода времени. Поэтому если мы зависим от чего-то абстрактного, вполне вероятно, что это не окажет на нас негативного влияния – эта вещь должна быть очень *стабильной*.

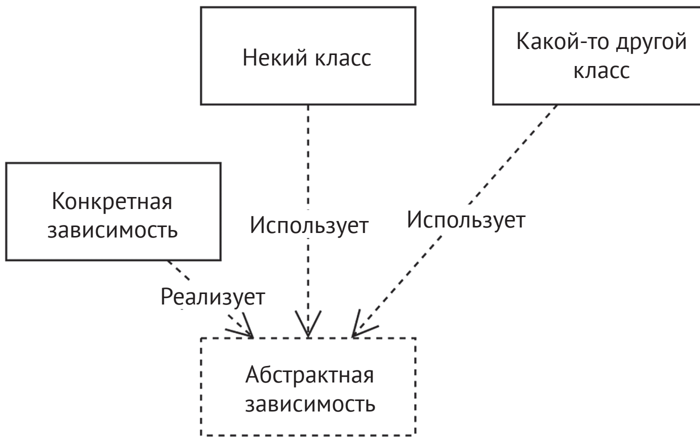


Рис. 11-2. Зависимость от абстрактных вещей

Вот здесь и встречаются обе концепции – устойчивость и абстрактность. Если мы рассматриваем устойчивость как вероятность того, что что-то изменится, тогда то, что справедливо для классов, справедливо и для пакетов. Как мы теперь знаем, лучше зависеть от стабильных пакетов, чем от нестабильных. Стабильные пакеты реже меняются, поэтому изменения зависимостей не будут оказывать негативного влияния на зависимый пакет. Точно так же безопаснее зависеть от абстрактных классов или интерфейсов, потому что они с меньшей вероятностью изменятся.

Можно следовать тем же рассуждениям, применяя понятие абстрактности к пакетам: было бы лучше зависеть от абстрактного пакета, чем от конкретного. По той же самой причине абстрактный пакет не будет содержать конкретных деталей реализации, которые могут быть подвержены изменениям. В течение более длительного периода времени он будет оставаться прежним.

Как определить, является ли пакет абстрактным

Вот в чем вопрос: можно ли пометить пакет как абстрактный или конкретный? Это возможно, даже если «быть абстрактным» не является логическим значением. Существует множество степеней абстрактности. Можно считать класс абстрактным, если он содержит *хотя бы один абстрактный метод*. Класс является конкретным, если у него нет абстрактных (или виртуальных) методов.

Абстрактность пакетов можно определить аналогичным образом. Пакет является абстрактным, если не содержит обычных классов, только интерфейсы и абстрактные классы. И пакет является конкретным, если имеет хотя бы один полностью реализованный, конкретный класс.

Тем не менее тут есть маленький нюанс. Согласно этому определению абстрактных и конкретных пакетов, пакет с 10 интерфейсами и 1 конкретным классом будет считаться конкретным пакетом, даже если содержит гораздо больше абстрактных вещей, нежели конкретных. Поэтому мы должны учитывать общее количество классов и интерфейсов.

А-метрика

Предлагаемый способ найти указание на абстрактность пакетов – вычислить количество абстрактных классов и интерфейсов в пакете, а затем разделить это число на общее количество конкретных классов, абстрактных классов и интерфейсов в этом пакете. Получившаяся вещь будет частным со значением где-то между 0 и 1. Мы называем это число А-метрикой пакетов:

```
A = C-abstract / (C-concrete + C-abstract)
```

Когда значение А-метрики пакета равно или близко к 0, это очень конкретный пакет. Он (почти) не содержит никаких интерфейсов, только конкретные классы, поэтому наполнен деталями реализации и, следовательно, может изменяться.

С другой стороны, когда А-метрика равна или близка к 1, это очень абстрактный пакет. Он не содержит (почти) никаких конкретных классов, но в основном это абстрактные классы и интерфейсы. Вполне вероятно, что эти абстрактные вещи со временем останутся прежними. В конце концов, только конкретные классы и, следовательно, конкретные пакеты могут изменяться.

Абстрактные вещи в стабильных пакетах

Итак, абстрактные пакеты тоже стабильны. Или, по крайней мере, должны быть таковыми. Вот где вступает в действие принцип *устойчивых абстракций*:

Пакеты, которые являются максимально стабильными, должны быть максимально абстрактными. Нестабильные пакеты должны быть конкретными. Абстракция пакета должна быть пропорциональна его стабильности[1].

Мы уже знаем, что пакет должен зависеть только от более стабильных пакетов. Теперь мы также знаем, что абстрактные вещи, вероятно, будут более стабильными, то есть они изменяются реже и менее резко. Следовательно, конкретные классы могут безопасно зависеть от них. Но что, если интерфейс является частью крайне нестабильного пакета? Тогда получается, что зависеть от этого пакета небезопасно. Нестабильный пакет, вероятно, будет меняться. Интерфейс наследует нестабильность содержащего его пакета.

Интерфейсам и абстрактным классам будет лучше в стабильном пакете. Стабильность самого пакета будет полезна для абстрактных вещей, которые в нем содержатся. В то же время абстрактные вещи полезны для стабильности пакета. Пакеты, которые применяют принцип *инверсии зависимостей*, начинают зависеть от него по причине абстрактных вещей, которые он содержит, что превращает его в более ответственный пакет и тем самым заставляет его стать более стабильным.

Верно и обратное: конкретным классам лучше в нестабильных пакетах.

Детали реализации классов, скорее всего, изменятся, и будет лучше, если бы это произошло в нестабильном пакете, который несет меньшую ответственность по отношению к зависимым пакетам. Однако если конкретный класс будет находиться в высокостабильном пакете, это сделает пакет более нестабильным, поскольку конкретный класс может измениться.

---

[1] *Robert C. Martin*. Engineering Notebook. C++ Report, Feb 1997 (PDF-версия доступна по адресу http://butunclebob.com/ArticleS.UncleBob.PrinciplesOfOod).

Абстрактность возрастает по мере роста устойчивости

Принцип *устойчивых абстракций* добавляет дополнительное требование. Он хочет объединить абстрактность и устойчивость в простое правило: пакет должен быть настолько абстрактным, насколько он стабилен.

Предположим, у вас есть несколько пакетов, и вы рассчитали для всех них значения I. Помните, что значение I указывает на устойчивость пакета: чем ближе значение к 1, тем более нестабилен пакет. Чем оно ближе к 0, тем оно стабильнее. Когда вы рисуете их на диаграмме, пакеты с наибольшим значением I находятся сверху, а те, у кого наименьшее значение I, находятся внизу. Когда вы переходите от пакета к его зависимостям, то сталкиваетесь с пакетами, чьи значения I только уменьшаются (*см*. рис. 11-3), то есть они становятся все более и более стабильными.

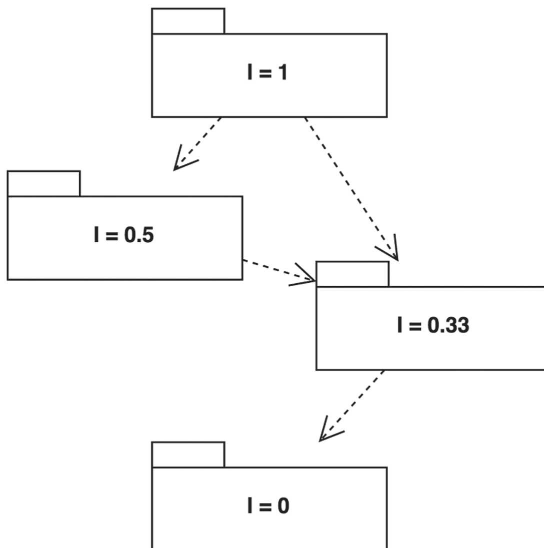


Рис. 11-3. Все зависимости направлены в сторону устойчивости

Чтобы выяснить, правильно ли вы применили принцип *устойчивых абстракций*, вам также необходимо вычислить значения A (путем деления числа абстрактных классов и интерфейсов на

общее количество классов и интерфейсов). Затем добавьте полученные значения A в диаграмму зависимостей (*см.* рис. 11-4).

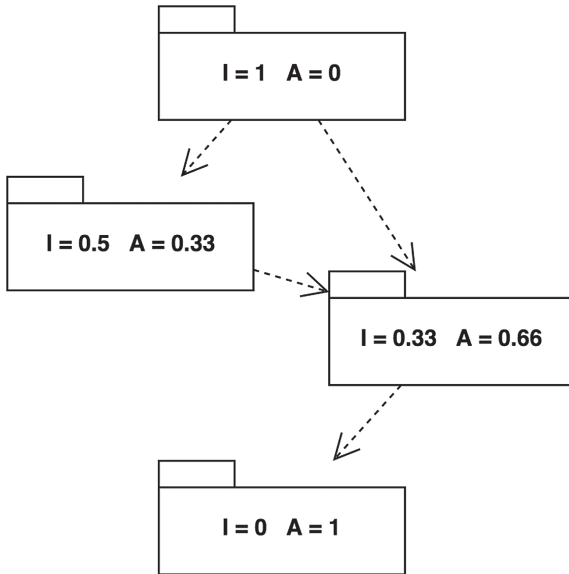


Рис. 11-4. Все зависимости направлены в сторону абстрактности

Теперь вам нужно только убедиться, что каждая стрелка зависимости ведет к пакету с более высоким значением A. Другими словами, зависимости должны иметь растущую абстрактность.

Строго говоря, значения I и A при сложении должны давать ровно 1. Это будет означать, что все пакеты *настолько абстрактны, насколько они стабильны*. Но это абсолютно нереально. Всегда есть некий запас. Тем не менее сумма I + A не должна быть слишком далека от 1. В целом очень абстрактные пакеты должны быть очень стабильными, а конкретные пакеты должны быть нестабильными.

Главная последовательность

Возможно, что стабильные пакеты содержат конкретные классы, а нестабильные пакеты – абстрактные классы или интерфейсы. Эти пакеты можно легко обнаружить, но существует много степеней устойчивости и абстрактности. Чтобы выяснить, какие пакеты имеют несбалансированные значения I и A, мы можем нанести

пакеты на диаграмму под названием *диаграмма главной последовательности.*

Сначала мы рисуем вертикальную ось, идущую от 0 до 1. Она обозначает степень *абстракции* пакета (выражена A). Затем мы рисуем горизонтальную ось, идущую от 0 до 1.

Она обозначает степень *нестабильности* пакета (выражена I). Наконец, мы рисуем диагональную линию от верхнего левого угла до нижнего правого угла. Эта линия называется *главной последовательностью* (*см.* рис. 11-5).

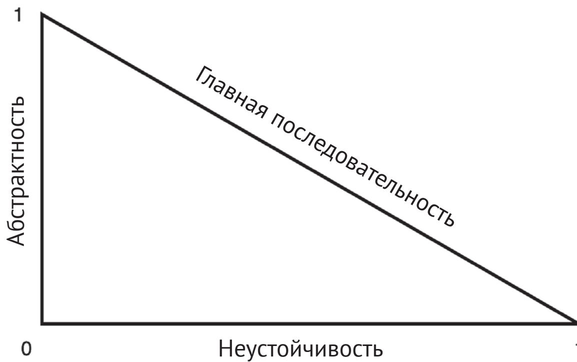


Рис. 11-5. Шаблон диаграммы главной последовательности

Теперь мы наносим каждый пакет в нужном месте на диаграмме, основываясь на его значениях I и A (*см.* рис. 11-6).

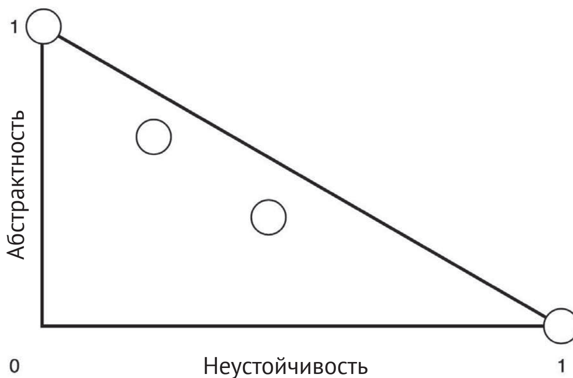


Рис. 11-6. Диаграмма главной последовательности для пакетов из предыдущей диаграммы зависимостей

Вы узнаете, что правильно применили как принцип *устойчивых зависимостей*, так и принцип *устойчивых абстракций*, если все пакеты будут находиться рядом или по диагонали, т. е. это главная последовательность. Согласно этому правилу, приведенная ранее диаграмма главной последовательности выглядит довольно хорошо.

Если вы обнаружите какой-либо пакет, который находится довольно далеко от главной последовательности, следует повнимательнее присмотреться к нему. Рассмотрите также окружающие его пакеты и попробуйте внести изменения в его зависимости или зависимые пакеты, чтобы добиться нужного уровня устойчивости.

Также может потребоваться изменить абстрактность пакета путем перемещения некоторых абстрактных классов или интерфейсов.

После исправления самых серьезных проблем, связанных с устойчивостью и абстракцией, вы должны регулярно возвращаться, чтобы посмотреть, не начали ли пакеты отклоняться от главной последовательности. Через некоторое время характер существующих пакетов может измениться из-за добавления в них новых функций, что в конечном итоге может привести к дисбалансу.

Типы пакетов

Когда вы пройдете по главной последовательности от верхнего левого угла к нижнему правому, то сначала столкнетесь с очень *конкретными, крайне нестабильными* пакетами. Все они должны быть пакетами прикладного уровня. Они наполнены деталями реализации, специфичными для фактического проекта, над которым вы работаете. Им разрешено быть конкретными, потому что от них не зависит никакой другой пакет. Следовательно, они являются нестабильными пакетами, которые должны меняться так же часто, как меняется бизнес.

Сделав еще несколько шагов по главной последовательности, вы найдете в пакетах посередине несколько абстрактные и несколько стабильные пакеты. На них гораздо меньше влияют внешние изменения, но в определенной степени им разрешено изменяться, не подвергая опасности весь проект.

Когда путешествие по главной последовательности закончится, вы окажетесь в царстве *абстрактных, стабильных пакетов*: основополагающих блоков вашего приложения. Они содержат множество интерфейсов и абстрактных классов, поэтому многие классы (а следовательно, и пакеты) зависят от них. Эти классы правильно при-

меняют принцип *инверсии зависимостей*, чтобы быть менее восприимчивыми к изменениям. Следовательно, эти абстрактные пакеты также должны быть стабильными. И они являются таковыми, потому что они ответственны и не имеют никаких зависимостей.

ИНТЕРФЕЙСНЫЕ ПАКЕТЫ

Если вы хотите создать высокоабстрактные стабильные пакеты, можно просто извлечь интерфейсы из существующих пакетов и поместить их все в один большой *интерфейсный пакет*.

Это не лучшее, что вы можете сделать для своего проекта. Поместив все интерфейсы в один пакет, вы нарушите принцип *совместного повторного использования*: некоторым клиентам требуется только один или два интерфейса из пакета, но они все равно должны зависеть от всего пакета.

Кроме того, не все интерфейсы должны зависеть от заданного пакета. У вас может быть много интерфейсов, предназначенных только для закрытого (private) использования в одном и том же пакете (в зависимости от вашего языка программирования вы можете пометить интерфейс как закрытый).

И последнее, о чем следует знать: прежде чем переносить интерфейсы в отдельный проект, убедитесь, что вы правильно применили принцип *разделения интерфейсов* ко всем из них. Таким образом, клиентам не придется зависеть от интерфейсов с методами, которые они не используют или не должны использовать.

Если вы применяете эти правила верно, можете создавать множество небольших интерфейсных пакетов, каждый из которых поддерживает одну специфическую функцию.

Странные пакеты

Что происходит в углах, которые находятся далеко от главной последовательности? Какие непослушные пакеты можно найти там (*см.* рис. 11-7)?

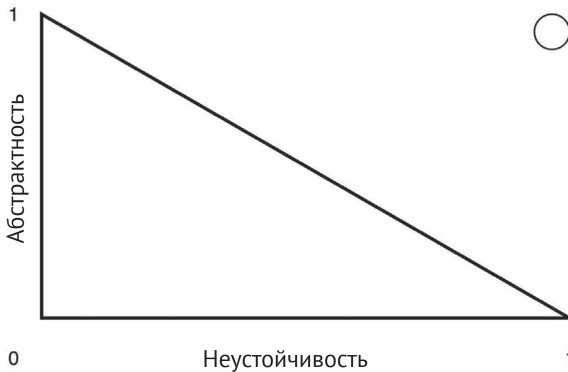


Рис. 11-7. Пакет в правом верхнем углу

В правом верхнем углу вы найдете как очень абстрактные, так и крайне нестабильные пакеты. Эти пакеты можно охарактеризовать как:

- неответственный (другие пакеты не зависят от них);
- зависимые (они зависят от множества других пакетов);
- абстрактные (они содержат только абстрактные классы или интерфейсы).

Это довольно странный тип пакета. Очень маловероятно, что такой пакет обнаружится в вашем проекте, поскольку он, скорее всего, будет содержать *мертвый код*. Ни в одной из частей проекта он так и не использовался, однако данный код является абстрактным, а это означает, что его нельзя использовать автономно – кто-то должен предоставить реализацию для него. Другими словами, подобные пакеты бесполезны, и нужно попытаться избавиться от них.

На противоположной стороне диаграммы, в нижнем левом углу (*см.* рис. 11-8), вы найдете пакеты, которые являются очень конкретными и в то же время достаточно стабильными.

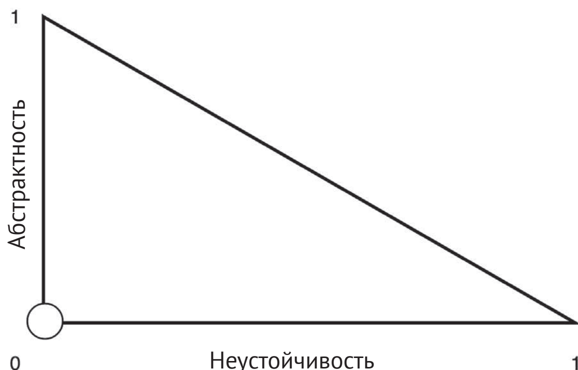


Рис. 11-8. Пакет в левом нижнем углу

Такие пакеты:

- ответственные (от них зависит множество пакетов);
- независимые (у них нет зависимостей);
- конкретные (они содержат только конкретные классы).

Этот тип пакета будет интенсивно использоваться на протяжении всего проекта. Поскольку для работы ему не нужны другие пакеты, возможно, это какая-то библиотека низкого уровня.

Тем не менее он не предлагает никаких абстракций для того, что делает. Это означает, что классам в других пакетах трудно естественным образом соответствовать принципу *инверсии зависимостей*: они должны зависеть от конкретных классов этого пакета, а не от интерфейсов.

Решение, позволяющее вернуть этот пакет снова в форму, состоит в том, чтобы применить принцип *инверсии зависимостей* к *зависимым* пакетам. Они больше не должны зависеть от конкретных классов из этого пакета, а должны зависеть от собственных интерфейсов или интерфейсов, определенных в каком-то более стабильном и абстрактном пакете.

Заключение

В этой главе мы обсудили тесную связь между устойчивостью и абстрактностью. Мы узнали, что чем стабильнее или устойчивее пакет, тем больше абстрактных вещей он должен содержать. Ана-

логом этого является то, что чем больше конкретных вещей содержит пакет, тем более нестабильным он становится.

Мы рассмотрели диаграмму главной последовательности, чтобы получить представление о различных типах пакетов и о том, где они находятся на скользящей шкале между конкретным/нестабильным и абстрактным/стабильным. Используя эту диаграмму, можно определить пакеты с необычными функциями.

Глава 12

Заключение

Теперь, когда мы обсудили так много принципов проектирования, вы, возможно, почувствовали, что пришло время стать немного более практичным. Вы просто хотите наконец-то начать создавать замечательные пакеты и делиться ими с коллегами или даже с теми, кто занимается разработкой открытого программного обеспечения по всему миру. Я определенно думаю, что вы должны это сделать. Перед этим я хочу дать вам несколько советов напоследок.

Создание пакетов – это сложно

В своей книге «Факты и ошибки разработки программного обеспечения» Роберт Гласс упоминает несколько интересных «фактов» касательно повторно используемых программных компонентов (или «пакетов»). Прежде всего:

> *Повторное использование «в малом» (библиотеки подпрограмм) [...] является хорошо решенной проблемой*[1].

Повторное использование «в малом»

Существуют всевозможные виды небольших общих свойств, которые очень полезны и помогают быстро решить постоянно возникающие проблемы, такие как сортировка массива, чтение байтов из файла и т. д. Они часто распространяются как «стандартные библиотеки» вместе со средой выполнения вашего языка программирования.

---

[1] *Robert L. Glass*. Facts and Fallacies of Software Engineering. Pearson Education, 2003.

Сочетая базовые функции, подобные этим, можно создавать некоторые новые функции низкого уровня. Пока эти функции достаточно малы и неспецифичны, проблем с их повторным использованием в других проектах быть не должно. Вам только нужно убедиться, что вы правильно выпустили код.

Повторное использование «в большом»

Когда мы продолжаем объединять и реструктурировать эти низкоуровневые свойства, мы в конечном итоге производим сложные и передовые программные компоненты. Если вы хотите повторно использовать целые программные компоненты в других проектах, это уже совсем другая история. Как говорит Гласс:

> *Повторное использование «в большом» (компоненты) по большей части остается нерешенной проблемой [...]*[1].

Гораздо сложнее подготовить эти более крупные компоненты для повторного использования, чем писать небольшие, многократно используемые функции. Основная причина состоит в том, что компоненты обычно выполняют специфическую для приложения цель. Большинство компонентов в проекте являются результатом фактических требований для этого проекта.

Несмотря на то что в мире по-прежнему существует множество проектов, которые удовлетворяют некоторым или большей части своих требований (возможно, потому, что они являются частью одной и той же предметной области бизнеса), ни одно из двух приложений не является одинаковым. Это означает, что всегда есть некий аспект компонента, который будет бесполезен в другом проекте или будет противоречить требованиям другого проекта и т. д.

Когда вы работаете с компонентом, который хотите сделать повторно используемым, может быть очень трудно, а может быть и невозможно учитывать все способы, которыми другие разработчики захотят его использовать. Часто вы видите, что существуют возможности для улучшения только после того, как кто-то указывает вам на это.

---

[1] *Robert L. Glass*. Facts and Fallacies of Software Engineering. Pearson Education, 2003.

Разнородность программного обеспечения

Тот факт, что повторное использование «в большом» настолько сложно, Гласс приписывает так называемой *разнородности программного обеспечения*:

> *Если, как многие подозревают, разнообразие приложений и предметных областей означает, что две проблемы не очень похожи друг на друга, то, вероятно, только те общие функции и задачи будут обобщены*[1].

Создание повторно используемых компонентов (или «пакетов») возможно только в том случае, если:

- предметная область двух проектов одинакова

или

- компонент предлагает функциональность общего назначения.

В первом случае компоненты моделируют некую часть общей предметной области с возможностью повторного использования, оставляя некоторые детали клиенту. Например, может быть возможно повторно использовать компонент онлайн-платежей в нескольких приложениях электронной коммерции. Или, если вы пишете программное обеспечение для управления полетами, как это делает НАСА, вы, вероятно, сможете повторно использовать некоторые компоненты в последующем проекте. Как упоминает Роберт Гласс, NASA сообщает, что количество повторного использования кода составляет около 70 %, что в основном следует объяснить тем фактом, что большинство проектов агентства имеют общую предметную область.

Во втором случае компоненты предлагают некие в целом полезные функции, такие как журналирование, общение с сервером по протоколу HTTP, обработка веб-запросов, преобразование Markdown в HTML и т. д. Эти вещи полезны в больших подмножествах всех приложений. Пока они предлагают множество вариантов конфигурации и правильно применяют принципы SOLID и разработки пакетов, эти компоненты или пакеты имеют хорошие шансы на повторное использование.

---

[1] *Robert L. Glass*. Facts and Fallacies of Software Engineering. Pearson Education, 2003.

Повторное использование компонентов возможно, но требует больше работы

Если вы хотите, чтобы ваш код можно было использовать многократно, вы должны постоянно заботиться о его расширяемости, читабельности, автоматических тестах, качестве кода и т. д. Если вы хотите, чтобы ваш повторно используемый код был успешным во множестве проектов, то должны знать о любых различиях в окружениях между этими проектами (разные версии языка программирования, разные операционные системы и т. д.). Вы должны обеспечить некий уровень заботы о пакете, например предлагая поддержку или исправляя ошибки. Нужно предложить некий опыт использования продукта, например написав документацию и предоставив примеры использования. Все усилия, приложенные вами, приводят к следующему практическому правилу, предложенному Глассом:

> *Компоненты многократного использования создать в три раза сложнее, чем одноразовые [...]*[1].

После всего этого вы все равно окажетесь в положении, когда вы – единственный, кто использовал компонент в проекте. Вы пока не знаете, как он будет вести себя в иных проектах, если он будет соответствовать ожиданиям других разработчиков. Поэтому было введено второе правило:

> *[...] повторно используемый компонент следует опробовать в трех разных приложениях, прежде чем он станет достаточно общим, чтобы принять его в библиотеку повторного использования*[2].

Создание пакетов – выполнимая задача

Я согласен с первым правилом. По моему опыту создание многократно используемого программного обеспечения занимает определенно больше времени, нежели создание программного обеспечения, которое таковым не является. Я не знаю точно, во сколько раз, но в три раза больше звучит правильно. Тем не менее вы, безу-

---

1 *Robert L. Glass*. Facts and Fallacies of Software Engineering. Pearson Education, 2003.

2 Там же.

словно, можете повлиять на это количество времени (и усилия), учитывая следующие факторы:

- количество функций, которые предоставляет ваш пакет;
- зона предметной области, охватываемая вашим пакетом;
- уровень расширяемости вашего пакета.

Уменьшение воздействия первого из трех правил

Если в вашем пакете слишком много функций, потому что вы хотите удовлетворить как можно большую группу пользователей, вероятно, вы будете тратить все больше и больше времени на обслуживание этого пакета. Все эти функции начнут мешать друг другу. Устранение ошибки в одной функции может нарушить работу другой функции. Кроме того, каждая из этих функций имеет свои зависимости, что делает пакет довольно нестабильным.

Если ваш пакет пытается охватить бóльшую зону предметной области, скорее всего, вы будете тратить много времени, пытаясь реализовать все мыслимые детали, которые *могут* быть частью этой области. Предметная область бизнеса обычно имеет так много аспектов, которые могут немного или даже в значительной степени отличаться для отдельных проектов в одной области, что невозможно охватить их все с помощью своего пакета. Пользователи вашего пакета всегда смогут указать на те детали, которые вы пропустили.

Эти два фактора, влияющих на количество времени и усилий, необходимых для создания пакета, связаны с *областью действия пакета*. Наш вывод, основанный на этих дискуссиях, должен заключаться в том, что *всегда нужно ограничивать область действия*.

Есть еще одна вещь, которую мы должны сделать, чтобы повлиять на количество времени, которое нам нужно инвестировать: мы должны убедиться, что наши пакеты очень расширяемы. Если пользователи пакета не могут изменить поведение его классов без изменения или переопределения фактического кода, они придут к вам, чтобы пожаловаться. Вместо того чтобы попросить *вас* добавить одну функцию, которой им недостает, вы должны разрешить им добавить эту функцию *самостоятельно*. Это должно сэкономить вам много времени и сделать ваш пакет очень привлекательным для пользователей.

Уменьшение воздействия второго из трех правил

Второе правило гласило, что мы должны опробовать наш повторно используемый компонент как минимум в трех разных приложениях. По моему опыту это не всегда необходимо. Хотя, соблюдать ли это правило или нет, зависит от ситуации.

Вы планируете поделиться этим кодом с миром? Тогда можете отправить его прямо сейчас и узнать, что произойдет. Конечно, это помогает представить потенциальные варианты использования и приспособить код, чтобы активировать их. Но всегда будет один крайний случай, о котором вы и не думали. Выпуск пакета с лицензией, которая не дает никаких гарантий пользователям, может быть отличным способом получить обратную связь, которая покажет вам, насколько ваш код в действительности можно использовать повторно.

Возможно, вы не собираетесь открывать исходный код, но хотите использовать его в других приложениях, созданных вами. Тогда вам не следует сразу помещать код в пакет. Просто скопируйте его в следующий проект и посмотрите, доказывает ли он свою полезность там.

Причиной для проведения этого различия является то, что при упаковке кода требуются значительные затраты. Как вы знаете, применение принципа *эквивалентности выпуска и повторного использования* к пакетам может дать много дополнительной работы. Вы еще не знаете, окупится ли она. Если вы использовали код только в одном проекте, вы пока не знаете, на самом ли деле он подходит для повторного использования. Сначала вам нужно выяснить, заслуживает ли код упаковки, посмотрев, насколько хорошо он адаптируется ко второму проекту. Вот почему вы не должны сразу же стремиться к повторному использованию любого фрагмента кода, который вы пишете для проекта. Только когда вы будете испытывать потребность в этом коде во втором проекте, а возможно, и в третьем, тогда вам следует рассмотреть возможность его упаковки.

Приложению обычно лучше подойдет код, специфичный для проекта и его предметной области. Всегда стремиться к абстрактным и универсальным решениям – интересное упражнение по программированию, но проекту и его разработчикам лучше иметь код, который распознает ожидаемое поведение приложения и концепции предметной области приложения. Вот почему не всегда

разумно писать код, держа в уме потенциальную возможность повторного использования.

Как только вы обнаружите пару классов, которые показывают некий потенциал для повторного использования, вы всегда можете переместить эти классы в отдельный каталог / пространство имен в рамках проекта и посмотреть, являются ли они жизнеспособными в качестве отдельного компонента. Это будет хорошим стимулом, чтобы улучшить дизайн этих классов и скрыть логику за *фасадом* или интерфейсом, предоставляемым компонентом. Однако на практике я обнаружил, что подавляющее большинство кода, который я писал таким образом, никогда не заканчивалось созданием реального пакета.

С другой стороны, глядя на собственный опыт разработчика пакетов, некоторые пакеты оказываются очень полезными для других. Мой личный возврат инвестиций с точки зрения затрат времени и сэкономленного времени не всегда такой благоприятный (я бы сказал – напротив). Тем не менее приятно видеть, что выпуск кода, который, по моему мнению, должен был быть общедоступным, действительно оказался полезным для других.

Создание пакетов – это легко?

В предыдущих разделах мы передумали и поменяли мнение «создавать пакеты сложно» на мнение, согласно которому «создавать пакеты возможно». Можно ли нам сделать следующий шаг и сделать вывод, что создавать пакеты легко? С одной стороны, да, я думаю, что это может быть очень легко. Когда вы привыкнете следовать правильной практике проектирования классов, писать чистый код и тестировать его до или во время его написания, вы обнаружите, что его нетрудно распространять в качестве пакета.

С другой стороны, если вы начнете с конкретного, единственного варианта использования кода, специфичного для проекта, и не проведете никаких измерений для обеспечения качества, будет очень сложно создать из него пакет для повторного использования. В этом случае я бы сказал, что вам все равно будет сложно поддерживать этот код.

В общем, мой совет – писать код как можно лучше. Таким образом, если вы когда-нибудь захотите использовать его повторно, то сможете достичь этой цели. А если вы этого не сделаете, то хорошо проведете время, поддерживая его. Пишите свой код правильно, и так всегда будет лучше.

Приложение А

Полный вариант класса Page

```
<?php
class Page
{
public $uri = null;
public $assigns = array();
public $page = array();
public $parent_node = 0;
public $site_title = '';
public $breadcrumbs = array();
public $auto_include_dir = '';
/* @public $smart Smarty */
public $smarty = null;
public $default_template = '';
public $template = '';
public $cms_login = null;
public $user_login = null;
public $available = true;
public $is_user = false;
public $is_admin = false;
public $menu_items = array();
public $caching = 1;
public $cache_id = null;
protected $_extra_request_parameters = array();
/**
* @param array $parameters
*/
public function setExtraRequestParameters(array $parameters)
{
$this->_extra_request_parameters = array_values($parameters);
}
/**
* @return array
*/
```

```
public function getExtraRequestParameters()
{
return $this->_extra_request_parameters;
}
public function __construct($uri)
{
$this->connect_db();
header('Content-Type: '.HEADER_CONTENT_TYPE);
$this->smarty = new Smarty;
if (isset($_GET['clear_cache']))
{
$this->smarty->clear_cache();
}
if (DEBUGGING)
{
$this->smarty->caching = false;
if (trusted_ip())
{
$this->smarty->debugging = true;
}
}
if (trusted_ip())
{
ini_set('display_errors', '1');
error_reporting(
E_ERROR | E_PARSE | E_WARNING
| E_USER_ERROR | E_USER_NOTICE | E_USER_WARNING);
}
else
{
$this->smarty->debugging = false;
ini_set('display_errors', '0');
error_reporting(0);
}
$this->smarty->template_dir = ROOT.'/site/templates';
$this->smarty->compile_dir = ROOT.'/site/templates_c';
$this->smarty->use_sub_dirs = true;
$this->default_template = DEFAULT_TEMPLATE;
if (!table_exists('content'))
{
require(ROOT.'/includes/install.php');
install();
}
$this->add_title_part(SITE_TITLE);
$this->cms_login = new LoginClass('admins', 'cms_login');
$this->user_login = new LoginClass('users', 'user_login');
if ($this->cms_login->isLoggedIn())
{
```

```
$this->is_admin = true;
}
if ($this->user_login->isLoggedIn())
{
$this->is_user = true;
}
$parsed_url = parse_url($uri);
$relative = ROOT;
$url = $parsed_url['path'];
$this->assign('header_content_type', HEADER_CONTENT_TYPE);
$this->determine_page($url);
if ($this->smarty->is_cached(", 'page_'.$this->
page['id']))
{
$this->smarty->display(
$this->default_template,
'page_'.$this->page['id']);
exit;
}
$this->smarty->register_function(
'translate',
'smarty_function_translate'
);
$this->smarty->register_modifier(
'translate',
'smarty_modifier_translate'
);
$this->smarty->register_function(
'url_for',
'smarty_function_url_for'
);
$this->open_page();
}
function get_page_info($id)
{
$result = mysql_query(
"SELECT c.id, c.uri, t.title, t.menu_name, " .
"c.node, c.skip_to_first_subpage FROM content c ".
"WHERE c.id='$id';"
);
if ($result && mysql_num_rows($result))
{
return mysql_fetch_assoc($result);
}
return false;
}
function determine_page($url)
{
```

```
$url_parts = explode('/', trim($url, '/'));
$page_ids = array();
$uri_prefix = '';
$parent_id = 0;
foreach($url_parts as $key => $part)
{
if ($key == 0 || empty($part))
{
unset($url_parts[$key]);
continue;
}
$result = mysql_query("SELECT id, skip_to_first_subpage ".
"FROM content c WHERE t.uri='".addslashes($part)."' ".
"AND c.parent_id='$parent_id';");
if (!$result)
{
throw new RuntimeException('MySQL error');
}
if (mysql_num_rows($result))
{
$page_id = mysql_fetch_assoc($result);
$parent_id = $page_id['id'];
$page_ids[] = $page_id;
unset($url_parts[$key]);
}
else
{
break;
}
}
// Оставшиеся фрагменты URL представляют собой дополнительные параметры
запроса;
$this->setExtraRequestParameters($url_parts);
while(empty($page_ids)
|| $page_ids[count($page_ids)-1]['skip_to_first_subpage'])
{
$page_id = $this->find_first_subpage(
$page_ids[count($page_ids)-1]['id']);
$result = mysql_query("SELECT id, skip_to_first_subpage ".
"FROM content WHERE id='$page_id';");
if ($result && mysql_num_rows($result))
$page_ids[] = mysql_fetch_assoc($result);
else
break;
}
$page = array();
foreach($page_ids as $id)
{
```

```
$page = $this->get_page_info($id['id']);
if ($page)
{
$uri_prefix .= '/'.$page['uri'];
$this->breadcrumbs[] = array('id' => $page['id'],
'uri' => $page['uri'], 'href' => $uri_prefix,
'menu_name' => $page['menu_name'],
'title' => $page['title']);
if ($page['node']) $this->parent_node = $page['id'];
$this->add_title_part($page['title']);
} else
break;
}
$this->page_url = $uri_prefix;
$this->page = $page;
$this->assign('breadcrumbs', $this->breadcrumbs);
return true;
}
function redirect($page_id)
{
if ($this->page['id'] != $page_id && $page_id > 0)
{
session_write_close();
header('HTTP/1.1 301 Moved Permanently');
header('Location: '.$this->get_url($page_id));
exit;
}
}
function open_page()
{
$result = mysql_query("SELECT * FROM content c ".
"WHERE id='{$this->page['id']}';");
if ($result && mysql_num_rows($result))
{
$this->page = mysql_fetch_assoc($result);
$this->page['contents'] = plain_text($this->page['contents']);
if ($this->is_admin)
{
if (!$this->page['available_for_admins'])
{
$this->page['contents'] =
TPL_NOT_AVAILABLE_FOR_ADMINS;
$this->available = false;
}
else if ($this->cms_login->user['id'] != 1
&& !$this->page['available_for_guests']
&& !$this->page['available_for_users']
&& !$this->has_permission(
```

```
$this->cms_login->user['id'],
$this->page['id'])
)
{
$this->page['contents'] =
TPL_NOT_AVAILABLE_FOR_SPECIFIC_ADMIN;
$this->available = false;
}
}
else if ($this->is_user)
{
if (!$this->page['available_for_users'])
{
$this->page['contents'] =
TPL_NOT_AVAILABLE_FOR_USERS;
$this->available = false;
}
}
else
{
if (!$this->page['available_for_guests'] && !$this->
is_user)
{
$this->page['contents'] =
TPL_NOT_AVAILABLE_FOR_GUESTS;
$this->available = false;
}
}
if (!$this->page['show_contents'])
{
$this->page['contents'] = TPL_INVISIBLE;
$this->available = false;
}
}
else
{
$this->page['contents'] = TPL_NOT_FOUND;
}
}
function has_permission($admin_id, $page_id)
{
$result = mysql_query("SELECT id FROM permissions WHERE ".
"admin_id='$admin_id' AND page_id='$page_id';");
if ($result && mysql_num_rows($result))
return true;
return false;
}
function find_first_subpage($parent_id = 0)
```

```
{
$result = mysql_query("SELECT id FROM content WHERE ".
"parent_id='$parent_id' ORDER BY priority ASC, id ASC;");
if ($result && mysql_num_rows($result))
return mysql_result($result, 0, 0);
}
function show_page()
{
if ($this->available)
{
$this->cache_id = 'page_'.$this->page['id'];
if ($this->page['include_file'] != "
&& file_exists(ROOT.'/site/'.$this->page['include_file']))
{
include_once(ROOT.'/site/'.$this->page['include_file']);
}
}
$this->menu = $this->load_menu();
$this->assign('menu', $this->menu);
$this->assign('page_id', $this->page['id']);
$this->assign('site_title', $this->
get_site_title());
$this->assign('contents', $this->page['contents']);
$this->assign('description', $this->page['description']);
$this->assign('keywords', $this->page['keywords']);
$this->assign('page_title', $this->page['title']);
$this->assign('page_url', $this->page_url);
$this->assign('subnavigation', $this->
subnavigation());
$this->assign('main_navigation', $this->
main_navigation());
$this->assign('is_admin', $this->is_admin);
$this->assign('is_user', $this->is_user);
$this->assign('parent_node', $this->parent_node);
$this->assign('meta_title', $this->
get_page_title(' - ', true));
$this->assign('timers', sfTimerManager::getTimers());
$this->smarty->caching = $this->caching;
$this->smarty->display(
($this->template != " ?
$this->template
: $this->default_template),
$this->cache_id);
}
function add_title_part($title_part)
{
$this->title_parts[] = $title_part;
}
```

```
function get_title_parts()
{
return $this->title_parts;
}
function get_page_title($separator = ' - ', $reverse = false)
{
$title_parts = $this->get_title_parts();
if ($reverse)
{
$title_parts = array_reverse($title_parts);
}
return implode($separator, $title_parts);
}
function menuitems($parent_id=0, $uri_prefix=")
{
$timer = sfTimerManager::getTimer('navigation');
$timer->startTimer();
$menu_items = array();
$result = mysql_query("SELECT c.id, t.uri, t.menu_name ".
"FROM content c "
"LEFT JOIN content_translations t ON c.id = t.content_id ".
"WHERE ".
"c.parent_id='$parent_id' AND c.show_in_menu='1' AND (".
($this->is_admin ? "c.available_for_admins='1' OR ":") .
($this->is_user ?
"c.available_for_users='1'"
: "c.available_for_guests='1'").
") ORDER BY c.priority ASC, c.id ASC;")
or $this->trigger_error(mysql_error());
if ($result && mysql_num_rows($result))
{
while ($item = mysql_fetch_assoc($result))
{
$item['href'] = $uri_prefix.$item['uri'];
$menu_items[$item['id']] = $item;
}
}
$timer->addTime();
return $menu_items;
}
function load_menu()
{
$menu = array();
$menu[0] = $this->menuitems(0, '/');
foreach($this->breadcrumbs as $item)
{
$menu[$item['id']] = $this->menuitems(
$item['id'],
```

```
$item['href'].'/'
);
}
return $menu;
}
function subnavigation()
{
if ($this->page['id'] != $this->parent_node
&& !empty($this->menu[$this->page['id']]))
return $this->menu[$this->page['id']];
else if ($this->page['parent_id'] != $this->>
parent_node
&& $this->page['parent_id'] != 0)
return $this->menu[$this->page['parent_id']];
}
function main_navigation()
{
return $this->menu[$this->parent_node];
}
function get_uri_prefix($page_id)
{
$uri_prefix = '/';
if ($page_id > 0)
{
foreach ($this->breadcrumbs as $item)
{
$uri_prefix .= $item['uri'].'/';
if ($item['id'] == $page_id) break;
}
}
return $uri_prefix;
}
public function get_url($page_id)
{
$timer = sfTimerManager::getTimer('get_url');
$timer->startTimer();
$url = ";
while ($page_id > 0)
{
$result = mysql_query("SELECT c.id, c.parent_id, t.uri ".
"FROM content c WHERE c.id='$page_id';");
if ($result && mysql_num_rows($result))
{
$page = mysql_fetch_assoc($result);
$url = '/'.$page['uri'].$url;
$page_id = $page['parent_id'];
}
else {
```

```
break;
}
}
$timer->addTime();
return $url;
}
function get_site_title()
{
$site_title = SITE_TITLE;
foreach($this->breadcrumbs as $crumb)
{
if ($crumb['id'] == $this->page['id'])
$crumb['title'] = $this->page['title'];
if ($crumb['title'] != ")
$site_title .= ' - '.$crumb['title'];
}
return $site_title;
}
public function connect_db()
{
$this->db_connection = @mysql_connect(
MYSQL_HOST,
MYSQL_USER,
MYSQL_PASSWORD);
if ($this->db_connection)
{
$this->db = @mysql_select_db(MYSQL_DB);
if (!$this->db)
{
?><p class="warning">Geen database.</p><?
exit;
}
}
else
{
?><p class="warning">Geen verbinding.</p><?
exit;
}
}
function assign($name, $value)
{
$this->smarty->assign($name, $value);
}
function trigger_error($message, $error_type=E_USER_WARNING)
{
$this->smarty->trigger_error($message, $error_type);
}
}
```

Предметный указатель

Абстрактность 35, 246
Абстракция 89, 101, 247
Диспетчер
  событий 97, 98, 218
Зацепление 18, 198, 199, 200
Независимый пакет 226
Нестабильный
  пакет 230, 242, 247
Ответственный
  пакет 227
Принцип единственной
  ответственности 25
Принцип
  открытости/закрытости 32
Принцип подстановки Барбары
  Лисков 50, 61, 69, 70
Связность 123
Фасад 111
Цепочка обязанностей 217, 218

L

Laravel 97, 98, 99, 100, 101, 102,
  103, 179, 189

R

README 146, 147, 148, 150

S

SOLID 16, 17, 22, 23, 83, 124, 125,
  152, 258
Symfony 12, 96, 99, 100, 101, 102,
  111, 145, 158, 160, 162, 186,
  188, 189

Z

Zend 189

Книги издательства «ДМК Пресс» можно заказать
в торгово-издательском холдинге «Планета Альянс» наложенным платежом,
выслав открытку или письмо по почтовому адресу:
115487, г. Москва, 2-й Нагатинский пр-д, д. 6А.
При оформлении заказа следует указать адрес (полностью),
по которому должны быть высланы книги;
фамилию, имя и отчество получателя.
Желательно также указать свой телефон и электронный адрес.
Эти книги вы можете заказать и в интернет-магазине: **www.a-planeta.ru**.
Оптовые закупки: тел. **(499) 782-38-89**.
Электронный адрес: **books@alians-kniga.ru**.

Маттиас Нобак

Принципы разработки программных пакетов:
Проектирование повторно используемых компонентов

Главный редактор *Мовчан Д. А.*
dmkpress@gmail.com
Перевод *Беликов Д. А.*
Корректор *Синяева Г. И.*
Верстка *Луценко С. В.*
Дизайн обложки *Мовчан А. Г.*

Формат 60×90 1/16.
Гарнитура «PT Serif». Печать цифровая.
Усл. печ. л. 17,13. Тираж 200 экз.

Веб-сайт издательства: www.dmkpress.com