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

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

Оглавление

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 вебсайта, 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);
}
}
```

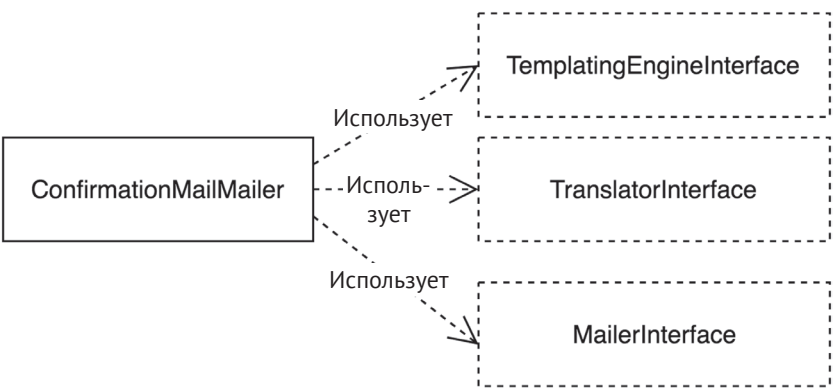

Рис. 1-1. Диаграмма исходной ситуации

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;
}
}
```

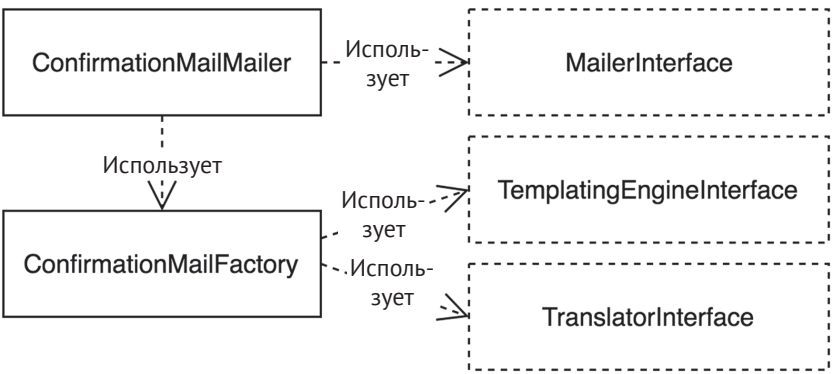

Рис. 1-2. Используем класс ConfirmationMailFactory

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.