The Addison-Wesley Signature Series

"Любой в состоянии написать код, который будет понятен компилятору. Хороший программист пишет код, который будет понятен человеку".

М. Фаулер, 1999 г.

A Martin Fowler Signature Book

Рефакторинг кода на JavaScript

Улучшение проекта существующего кода

Мартин Фаулер

при участии

Кента Бека

Диалектика

Второе издание

Список рефакторингов

Рефакторинг
кода на JavaScript

Улучшение проекта существующего кода

Refactoring

Improving the Design of Existing Code

Second Edition

Martin Fowler
with contributions by Kent Beck

Addison-Wesley

Boston • Columbus • New York • San Francisco • Amsterdam • Cape Town
Dubai • London • Madrid • Milan • Munich • Paris • Montreal • Toronto • Delhi • Mexico City
São Paulo • Sydney • Hong Kong • Seoul • Singapore • Taipei • Tokyo

Рефакторинг кода на JavaScript

Улучшение проекта существующего кода

Второе издание

Мартин Фаулер
при участии Кента Бека

Москва · Санкт-Петербург
2019

ББК 32.973.26-018.2.75
Ф28
УДК 681.3.07

ООО "Диалектика"
Зав. редакцией *С.Н. Тригуб*

Перевод с английского и редакция канд. техн. наук *И.В. Красикова*

По общим вопросам обращайтесь в издательство "Диалектика" по адресу:
info@dialektika.com, http://www.dialektika.com

Фаулер, Мартин.

Ф28 Рефакторинг кода на JavaScript: улучшение проекта существующего кода, 2-е изд. : Пер. с англ. — СПб. : ООО "Диалектика", 2019. — 464 с. : ил. — Парал. тит. англ.

ISBN 978-5-907144-59-0 (рус.)

ББК 32.973.26-018.2.75

Научно-популярное издание

Мартин Фаулер

Рефакторинг кода на JavaScript: улучшение проекта существующего кода 2-е издание

Подписано в печать 12.07.2019. Формат 70x100/16.
Гарнитура Times.
Усл. печ. л. 29,0. Уч.-изд. л. 20,0.
Тираж 500 экз. Заказ № 5984.

Отпечатано в АО "Первая Образцовая типография"
Филиал "Чеховский Печатный Двор"
142300, Московская область, г. Чехов, ул. Полиграфистов, д. 1
Сайт: www.chpd.ru, E-mail: sales@chpd.ru, тел. 8 (499) 270-73-59

ООО "Диалектика", 195027, Санкт-Петербург, Магнитогорская ул., д. 30, лит. А, пом. 848

ISBN 978-5-907144-59-0 (рус.)
ISBN 978-0-13-475759-9 (англ.)

Оглавление

Содержание

Посвящается Синди
— Мартин

Предисловие к первому изданию

Понятие рефакторинга возникло в кругах, близких к Smalltalk, но очень быстро проложило дорогу к другим языкам программирования. Поскольку оптимизация является неотъемлемой частью развития программного обеспечения, этот термин появляется, как только проектировщики начинают вести профессиональные беседы. Он используется и когда речь идет об уточнении иерархии классов, и когда обсуждается, сколько строк кода можно удалить. Программисты знают, что с первого раза хорошо работающую программу не получить — она должна развиваться постепенно, по мере накопления опыта. Они также знают, что код будет куда чаще читаться и изменяться, чем писаться "с нуля". Рефакторинг является ключом к поддержке удобочитаемости и изменяемости кода — как для всего программного обеспечения в целом, так и для конкретных программ.

В чем же заключается проблема? Ответ прост: рефакторинг — это риск. Он требует изменения рабочего кода, в ходе которого могут возникнуть не только улучшения, но и новые ошибки. Небрежно выполненный рефакторинг может отбросить вас назад на дни и даже недели. Еще более рискованным оказывается рефакторинг, выполняемый неофициально или от случая к случаю. Вы начинаете копаться в коде. Вскоре вы обнаруживаете новые возможности для внесения изменений и закапываетесь глубже. Чем глубже вы копаете, тем больше возможностей для изменений обнаруживаете. В конце концов вы выкапываете яму, из которой уже не в состоянии выбраться. Чтобы эта яма не превратилась в могилу, рефакторинг необходимо выполнять систематически. Когда я со своими соавторами написал книгу "Design Patterns"[1], в ней было упомянуто, что проектные шаблоны обеспечивают цели для рефакторинга. Однако определение цели является лишь частью проблемы; другой задачей является преобразование кода, позволяющее достичь поставленной цели.

Мартин Фаулер (Martin Fowler) и его соавторы внесли неоценимый вклад в развитие объектно-ориентированного программного обеспечения, пролив свет на процесс рефакторинга. В этой книге описаны принципы и передовой опыт рефакторинга и указано, когда и где следует начинать работу с кодом для его

---

[1] Имеется русский перевод: Гамма Э., Хелм Р., Джонсон Р., Влиссидес Д. *Приемы объектно-ориентированного проектирования. Паттерны проектирования.* — С-Пб, "Питер", 2000. — *Примеч. пер.*

улучшения. В основе книги находится подробный перечень рефакторингов. Каждый рефакторинг описывает мотивацию и технологию преобразования кода. Некоторые из разновидностей рефакторинга, такие как, например, "Извлечение метода" или "Перемещение поля", могут показаться очевидными. Но не обманывайтесь — понимание технологии таких рефакторингов является ключом к выполнению организованного рефакторинга. Описания рефакторингов в этой книге помогут вам изменить свой код небольшими порциями за один раз, снижая тем самым риск резкого изменения самих принципов вашего проекта. Названия этих рефакторингов очень быстро займут свое место в вашем словарном запасе.

Первый опыт "пошагового" рефакторинга я получил, работая на пару с Кентом Беком (Kent Beck). Он обеспечил поэтапное применение описанных в этой книге рефакторингов, поразив меня эффективностью своих действий. Я не только получил более надежный код, но и почувствовал себя спокойнее. Настоятельно рекомендую вам испытать эти рефакторинги: и ваш код, и вы сами почувствуете себя намного лучше.

— Эрих Гамма (Erich Gamma)
Object Technology International, Inc.
Январь 1999 г.

Введение

Как-то раз один консультант знакомился с проектом, связанным с разработкой программного обеспечения. Он посмотрел часть готового кода, который был основан на некоторой иерархии классов. Пытаясь пробраться сквозь нее, консультант нашел ее слишком запутанной. Классы в основе иерархии использовали ряд предположений о том, как будут действовать другие классы, и эти предположения были реализованы в коде производных классов. Однако данный код подходил не для всех подклассов и в результате нередко в них перекрывался. Сократить необходимость в перекрытии кода можно было бы путем внесения небольших изменений в суперкласс. В некоторых местах выполнялось дублирование поведения суперкласса, связанное с тем, что назначение суперкласса было недостаточно очевидным. Возникали также ситуации, когда ряд подклассов реализовывал одно и то же поведение, которое можно было бы безболезненно переместить выше в иерархии классов.

Рекомендация консультанта заключалась в том, чтобы пересмотреть и подчистить код. Конечно же, это не вызвало радости у руководства. Ведь программа была вполне работоспособной, а сроки поджимали. Поэтому было принято обычное решение — заняться рефакторингом потом. Когда-нибудь. Может быть...

Консультант обратился также к программистам, указав им на найденные недостатки. Программисты отнеслись к замечаниям иначе. По сути, они даже не были виновны — зачастую, чтобы заметить проблему, нужен взгляд со стороны. Потратив пару дней на переделку иерархии, программисты сумели сократить код вдвое без снижения функциональности системы. Результат их вполне удовлетворил, тем более что в итоге облегчилось добавление новых классов в иерархию, а также использование уже имеющихся классов в других местах системы.

Такое поведение программистов вызвало неприятие у руководства. График требовал ускорения работ, а эти два дня, затраченные программистами, не добавили ни одной из функциональностей, которых все еще не хватало в системе. Уже имевшийся код работал — просто он был немного менее понятным и "чистым". Руководство можно понять. Коммерческий результат приносит работающий код, а не код, который нравится "академической крысе". Консультант же предложил модифицировать и другие части кода, что могло приостановить работы на пару недель, и лишь ради более красивого кода, а не для расширения его возможностей.

Что вы скажете об этой истории? Кто, по-вашему, был прав? Может, действительно права старая программистская мудрость "Не трогай работающую систему"?

Думаю, вы сразу догадались, что этим консультантом был я. Шестью месяцами позже проект бесславно закрылся, рухнув под собственной тяжестью — код стал слишком громоздким для отладки или получения приемлемой производительности.

Чтобы возобновить проект, был привлечен другой консультант — Кент Бек, и практически все пришлось писать заново. Кент спроектировал многое совершенно иначе, чем ранее, при этом настояв на постоянном совершенствовании кода с применением рефакторинга. Успех описанного проекта и роль рефакторинга в этом успехе подвигли меня на создание данной книги. Книга позволила мне поделиться опытом и знаниями, приобретенными Кентом и другими специалистами при использовании рефакторинга для повышения качества программного обеспечения.

С тех пор рефакторинг стал признанной частью словаря программирования, а первое издание этой книги стало бестселлером. Однако восемнадцать лет для книги по программированию — это глубокая старость, и я постоянно чувствовал, что пришло время вернуться и переработать ее. С одной стороны, это заставило меня переписать многие страницы; с другой — в книге мало что изменилось. Суть рефакторинга та же; большинство ключевых рефакторингов остаются по существу такими же. Но я надеюсь, что новая книга поможет научиться эффективно проводить рефакторинг еще многим программистам.

Что такое рефакторинг

Рефакторинг — это процесс изменения программной системы, в ходе которого внешнее поведение кода остается неизменным при усовершенствовании его внутренней структуры. Это систематизированный способ очистки кода, минимизирующий возможность появления новых ошибок. По сути, рефакторинг кода представляет собой улучшение проекта уже после того, как этот код написан.

"Улучшение проекта после написания кода" звучит непривычно. При нынешнем понимании процесса разработки программного обеспечения мы сначала создаем проект, а потом пишем код. Первым шагом идет проектирование, а уже затем кодирование. Со временем код будет изменяться, а целостность системы и ее соответствие первоначальному проекту постепенно размываться. Кодирование понемногу перестает быть инженерным искусством и превращается в хакерство.

Рефакторинг же представляет собой нечто противоположное. Он позволяет, взяв плохой и беспорядочный проект, превратить его в ясно структурированный код. Каждый шаг этого преобразования чрезвычайно прост. Это может быть перемещение поля из одного класса в другой, выделение части исходного текста из метода и ее перемещение в отдельный метод, перемещение некоторых фрагментов кода в том или ином направлении иерархии классов. Кумулятивный эффект таких малозаметных изменений может привести к существенному улучшению

программы. Этот процесс оказывается прямой противоположностью описанной выше тенденции постепенной деградации программного проекта.

При рефакторинге изменяется баланс между разными этапами работ. Проектирование становится не отдельным начальным этапом разработки, а непрерывным процессом. В ходе работы над проектом вы постоянно рассматриваете возможность его улучшения. В результате получается программный проект, качество которого остается высоким все время работы над проектом.

О чем эта книга

Эта книга представляет собой руководство по рефакторингу; она написана для программистов-профессионалов. Моя цель при написании книги — показать, как выполнять рефакторинг управляемо и эффективно. Вы научитесь выполнять рефакторинг так, чтобы не вносить при этом в код новые ошибки, а постоянно улучшать его структуру.

Обычно книги начинаются с введения. Хотя я согласен с этой традицией, мне кажется слишком сложным начинать знакомство с рефакторингом с общего обсуждения или определений. Поэтому я начну с примера. В главе 1, "Первый пример рефакторинга", приводится небольшая программа с обычными недостатками, которая с помощью рефакторинга превращается в более приемлемую объектно-ориентированную программу. Попутно мы рассмотрим как процесс рефакторинга в целом, так и применение некоторых полезных рефакторингов. Это ключевая глава для понимания того, чем в действительности является рефакторинг.

В главе 2, "Принципы рефакторинга", я рассказываю об общих принципах рефакторинга более детально; там же я даю некоторые определения и описываю основные причины для проведения рефакторинга. Здесь же рассматриваются и некоторые проблемы, связанные с рефакторингом. В главе 3, "Запах в коде", посвященной тому, как найти запах в коде и избавиться от него, мне помогает Кент Бек. Очень большую роль при выполнении рефакторинга играет тестирование, поэтому глава 4, "Создание тестов", учит встраивать тесты в код.

Основной материал книги, который представляет собой каталог методов рефакторинга, занимает все остальные главы книги. Хотя это ни в коем случае не исчерпывающий каталог, он охватывает ключевые рефакторинги, которые могут понадобиться большинству разработчиков. Он вырос из заметок, которые я сделал, когда узнал о рефакторинге в конце 1990-х годов, и я до сих пор использую эти заметки, так как не могу помнить всё. Когда я хочу что-то сделать, например выполнить рефакторинг *Разделение этапа* (с. 201), каталог напоминает мне, как сделать это безопасно, по одному шагу. Я надеюсь, что это та часть книги, к которой вы будете часто возвращаться.

Примеры кода на JavaScript

Как и в большинстве технических областей разработки программного обеспечения, для иллюстрации концепций очень важны примеры кода. Однако рефакторинги на разных языках выглядят в основном одинаково. Иногда имеются определенные тонкости, на которые язык заставляет обращать внимание, но основные элементы рефакторинга остаются общими.

Я выбрал для иллюстрации рефакторингов JavaScript, так как чувствовал, что этот язык будет понятен и доступен для большинства читателей. Для вас не должно составлять проблемы адаптировать рефакторинг к тому языку, которым вы пользуетесь в настоящее время. Я стараюсь не задействовать какие-либо сложные части языка, поэтому вам должно быть достаточно поверхностного знания JavaScript, чтобы понимать суть рефакторинга. Мое применение JavaScript, конечно, не следует считать рекламой или рекомендацией этого языка.

Хотя я использую JavaScript в своих примерах, это не означает, что применимость методов в этой книге ограничена JavaScript. В первом издании использовался язык программирования Java, и многие программисты сочли книгу полезной, хотя и не написали ни одного класса на Java. Я продемонстрировал было общность методов, используя для примеров дюжину разных языков, но почувствовал, что это будет слишком запутывать читателя. Тем не менее эта книга написана для программистов, пишущих на любом языке. За исключением разделов с примерами, я не делаю никаких предположений о языке. Я ожидаю, что читатель усвоит мои общие комментарии и применит их к языку, который использует в повседневной работе, адаптировав соответствующим образом примеры на JavaScript.

Это означает, что, кроме обсуждения конкретных примеров, когда я говорю о "классе", "модуле", "функции" и тому подобном, я использую эти термины в общем смысле, а не в качестве конкретных терминов модели языка JavaScript.

Тот факт, что я выбрал в качестве языка примеров JavaScript, означает также, что я стараюсь избегать применения стилей JavaScript, которые будут менее знакомы тем, кто не является программистом на JavaScript. Это не книга по рефакторингу в JavaScript, а книга по рефакторингу, в которой в качестве примеров используется JavaScript. Существует много интересных рефакторингов, специфичных для JavaScript, но они выходят за рамки данной книги.

На кого рассчитана эта книга

Данная книга предназначена для профессиональных программистов. В примерах и обсуждении имеется множество кода, который нужно прочесть и понять. Все примеры написаны на языке программирования JavaScript, но должны быть

применимы для большинства языков программирования. Я ожидаю, что у читателя имеется некоторый опыт, чтобы понять то, что излагается в книге, но не предполагаю наличия у него обширных знаний.

Хотя предполагается, что основным читателем этой книги будет разработчик, стремящийся изучить рефакторинг, эта книга полезна и для тех, кто уже понимает, что такое рефакторинг. Я старался объяснить, как работают различные рефакторинги, чтобы опытный разработчик мог использовать этот материал в качестве учебного для наставничества по отношению к своим менее опытным коллегам.

Хотя рефакторинг ориентирован на исходные тексты, он существенно влияет и на проектирование. Руководителям-проектировщикам и архитекторам жизненно необходимо понимать принципы рефакторинга и использовать их в своих проектах. Пожалуй, будет лучше, если рефакторингом будет руководить опытный и уважаемый человек. Он сможет лучше понять принципы, на которых основан рефакторинг, и адаптировать их для конкретного проекта. Это особенно важно в случае языков программирования, отличных от JavaScript, так как приведенные в книге примеры потребуют определенной адаптации.

Можно получить пользу от книги, даже не читая ее полностью.

- **Если вы хотите понять, что такое рефакторинг**, достаточно прочесть главу 1, "Первый пример рефакторинга"; приведенный пример должен облегчить понимание.
- **Если вы хотите понять, для чего нужен рефакторинг**, прочтите две первые главы. В них рассказывается о том, что такое рефакторинг и почему он необходим.
- **Если вы хотите узнать, когда следует применять рефакторинг**, прочтите главу 3, "Запах в коде". В ней описаны признаки, говорящие о необходимости применения рефакторинга.
- **Если вы хотите выполнить рефакторинг**, прочтите первые четыре главы полностью. Затем пропустите все вплоть до каталога рефакторингов. Нет необходимости вдаваться во все детали. Когда вам понадобится какой-либо из методов рефакторинга, вы сможете прочесть детальную информацию о нем и использовать ее в своей работе. Каталог представляет собой справочный раздел, так что читать все подряд необязательно.

Важным вопросом при написании этой книги были названия различных рефакторингов. Хорошая терминология помогает нам общаться, поэтому, когда один разработчик советует другому извлечь некоторый код в функцию или разбить некоторые вычисления на отдельные фазы, оба понимают, что речь идет о рефакторингах *Извлечение функции* (с. 152) и *Разделение этапа* (с. 201). Эти названия помогают и при выборе автоматического рефакторинга.

На плечах других

Хочу сразу же сообщить, что эта книга обязана своим появлением тем, чья работа в 1990-х годах позволила развить область рефакторинга. Их опыт вдохновил меня на то, чтобы написать первый вариант этой книги, и, хотя прошло много лет, я продолжаю ценить тот фундамент, который они заложили. В идеале такую книгу должен был написать кто-то из них, но время и энергия для этой работы нашлись у меня.

Двумя ведущими сторонниками рефакторинга являются Уорд Каннингем (Ward Cunningham) и Кент Бек (Kent Beck). Для них рефакторинг давно стал центральной частью процесса разработки, принятой для ежедневного применения с целью получения всех его преимуществ. Именно сотрудничество с Кентом показало мне всю важность рефакторинга и подвигло меня на написание этой книги.

Ральф Джонсон (Ralph Johnson) возглавляет группу в Университете штата Иллинойс в Урбана-Шампань, известную своим вкладом в объектную технологию. Ральф давно является сторонником рефакторинга, как и множество его студентов. Автором первой работы, посвященной рефакторингу, стал Билл Опдайк (Bill Opdyke) со своей докторской диссертацией. Джон Брант (John Brant) и Дон Робертс (Don Roberts) не ограничились словами и создали инструментарий — Refactoring Browser, — предназначенный для выполнения рефакторинга исходных текстов на языке Smalltalk.

Со времени первого издания этой книги рефакторинг получил существенное развитие. В частности, огромный вклад в облегчение жизни программистов внесла работа тех, кто добавил автоматизированные рефакторинги в инструменты разработки. Легко считать само собой разумеющимся, что можно переименовать широко используемую функцию с помощью простой последовательности нажатий клавиш, но эта легкость обеспечена тяжелыми усилиями команд программистов, работавших над интегрированными средами разработки, работа которых так помогает нам всем.

Благодарности

Несмотря на использование результатов упомянутых выше исследований, для написания книги мне понадобилась большая помощь. В первую очередь, она была оказана мне Кентом Беком. Он познакомил меня с рефакторингом и именно под его влиянием я начал собирать материал по рефакторингу. Кент помог мне концепцией "кода с запахом" (code smells). Я часто думаю, что сам бы он смог написать эту книгу лучше меня, но, как я уже говорил, время для этой работы нашлось у меня, и мне остается только надеяться, что мой труд не напрасен.

Все авторы технических книг, которых я знаю, упоминают о большой благодарности техническим рецензентам. Мы все пишем работы, полные недочетов, которые замечают наши коллеги, выступающие в качестве рецензентов. Я сам не занимаюсь технической проверкой (отчасти потому, что не думаю, что у меня это хорошо получится) и потому восхищаюсь теми, кто делает это так качественно. Чтение чужой книги — дело не прибыльное, так что эта работа должна рассматриваться как великий акт щедрости.

Когда я начал серьезную работу над книгой, я сформировал список рассылки для своих советчиков, которым отправлял черновики новых материалов и просил их высказать свое мнение. Я хочу поблагодарить за отзывы в этом списке рассылки следующих участников: Арло Белши (Arlo Belshee), Авди Гримм (Avdi Grimm), Бет Андерс-Бек (Beth Anders-Beck), Билл Уэйк (Bill Wake), Брайан Гатри (Brian Guthrie), Брайан Марик (Brian Marick), Чад Уатингтон (Chad Wathington), Дейв Фарли (Dave Farley), Дэвид Райс (David Rice), Дон Робертс (Don Roberts), Фред Джордж (Fred George), Джайлс Александер (Giles Alexander), Грег Донч (Greg Doench), Хьюго Корбуччи (Hugo Corbucci), Айвен Мур (Ivan Moore), Джеймс Шор (James Shore), Джей Филдс (Jay Fields), Джессика Керр (Jessica Kerr), Джошуа Кериевски (Joshua Kerievsky), Кевлин Хенни (Kevlin Henney), Лучано Рамалхо (Luciano Ramalho), Маркос Бризено (Marcos Brizeno), Майкл Фезерс (Michael Feathers), Патрик Куа (Patrick Kua), Пит Ходжсон (Pete Hodgson), Ребекка Парсонс (Rebecca Parsons) и Триша Джи (Trisha Gee).

В этой группе я хотел бы особо отметить помощь, полученную по JavaScript от Бет Андерс-Бек, Джеймса Шора и Пита Ходжсона.

После написания первой законченной версии черновика я отправил его на новое рецензирование уже как единое целое и получил невероятно подробные комментарии от Уильяма Чаргина (William Chargin) и Майкла Хангера (Michael Hunger). Я также получил много полезных комментариев от Боба Мартина (Bob Martin) и Скотта Дэвиса (Scott Davis). Свой вклад внес и Билл Уэйк (Bill Wake), сделавший полный обзор первой версии черновика.

Мои коллеги по ThoughtWorks являются постоянным источником идей и отзывов о моих работах. Их бесчисленное множество вопросов, комментариев и наблюдений подпитывали мое мышление при написании этой книги. Одна из замечательных особенностей работы в ThoughtWorks заключается в том, что мне позволяют тратить на написание статей значительное время. В частности, я очень ценю регулярные обсуждения и идеи Ребекки Парсонс (Rebecca Parsons), нашего технического директора.

Моим редактором в издательстве Pearson является Грег Донч (Greg Doench), справлявшийся со множеством вопросов при подготовке книги к публикации. Мне было приятно работать с производственным редактором Джули Нахиль (Julie Nahil), а также с техническими редакторами Дмитрием Кирсановым (Dmitry Kirsanov) и Алиной Кирсановой (Alina Kirsanova).

Ждем ваших отзывов!

Вы, читатель этой книги, и есть главный ее критик. Мы ценим ваше мнение и хотим знать, что было сделано нами правильно, что можно было сделать лучше и что еще вы хотели бы увидеть изданным нами. Нам интересны любые ваши замечания в наш адрес.

Мы ждем ваших комментариев и надеемся на них. Вы можете прислать нам бумажное или электронное письмо либо просто посетить наш веб-сайт и оставить свои замечания там. Одним словом, любым удобным для вас способом дайте нам знать, нравится ли вам эта книга, а также выскажите свое мнение о том, как сделать наши книги более интересными для вас.

Отправляя письмо или сообщение, не забудьте указать название книги и ее авторов, а также свой обратный адрес. Мы внимательно ознакомимся с вашим мнением и обязательно учтем его при отборе и подготовке к изданию новых книг.

Наши электронные адреса:

E-mail: `info@williamspublishing.com`
WWW: `http://www.williamspublishing.com`

Глава 1

Первый пример рефакторинга

С чего же начать описание рефакторинга? Традиционно изложение новой концепции начинается с рассказа об исторических корнях, пояснения базовых принципов и т.п. Но когда на очередной конференции я слышу такое вступление, то сразу засыпаю. Мой мозг начинает работать в фоновом режиме с низким приоритетом, периодически опрашивая окружающую обстановку — не начали ли приводить конкретные примеры.

Тут я просыпаюсь, поскольку примеры помогают понять происходящее. Принципы позволяют легко делать обобщения, но, опираясь на них, очень тяжело понять, как применять их на практике. А пример все проясняет.

Поэтому начинаю свою книгу с примера рефакторинга. Пока я буду о нем рассказывать, вы узнаете, как действует рефакторинг, и получите представление о его выполнении. После этого я смогу дать обычное снотворное введение, знакомящее с историей и основными идеями и принципами.

Однако с упомянутым примером у меня возникли серьезные проблемы. Если взять в качестве примера большую программу, читателю будет трудно вникнуть в ход рефакторинга. (Я пытался поступить таким образом и выяснил, что не слишком сложный пример занял более сотни страниц текста.) Если же остановиться на небольшой программе, простой для понимания, то сколь-нибудь полно показать преимущества рефакторинга не получится.

Так я столкнулся с классической проблемой всех, кто пытается описывать технологии, полезные для программ из реальной жизни. Честно говоря, рефакторинг, который я покажу, не стоит затраченных на него усилий — такая маленькая программа будет использована в качестве примера. Но если приведенный код является частью куда большей системы, то рефакторинг сразу оказывается очень важным. Так что при изучении примера представьте его себе в контексте гораздо большей системы.

Начальная точка

В первом издании этой книги моя первая программа выводила счет из пункта видеопроката — что теперь может заставить многих из вас недоуменно спросить: "Что такое видеопрокат?" Вместо того чтобы отвечать на этот вопрос, я изменил свой пример.

Представьте себе театральную компанию, актеры которой дают представления на различных мероприятиях. Как правило, клиент запрашивает несколько постановок, и компания взимает плату в зависимости от размера аудитории и вида пьесы, которую они ставят. В настоящее время компания может представлять два вида пьес: трагедии и комедии. Вместе со счетом за услуги компания предоставляет клиентам бонусы, которые они могут использовать для скидок при будущих заказах, — эдакий механизм оплаты лояльности клиентов.

Исполнители хранят данные о своих пьесах в простом файле в формате JSON, который выглядит примерно так.

plays.json...

```
{
  "hamlet":  {"name": "Hamlet", "type": "tragedy"},
  "as-like": {"name": "As You Like It", "type": "comedy"},
  "othello": {"name": "Othello", "type": "tragedy"}
}
```

Данные для выставления счетов также находятся в JSON-файле.

invoices.json...

```
[
  {
    "customer": "BigCo",
    "performances": [
      {
        "playID": "hamlet",
        "audience": 55
      },
      {
        "playID": "as-like",
        "audience": 35
      },
      {
        "playID": "othello",
        "audience": 40
      }
    ]
  }
]
```

Код, который печатает счет, представляет собой простую функцию:

```
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;

  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                       { style: "currency", currency: "USD",
                         minimumFractionDigits: 2 }).format;

  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = 0;

    switch (play.type) {
      case "tragedy":
        thisAmount = 40000;

        if (perf.audience > 30) {
          thisAmount += 1000 * (perf.audience - 30);
        }

        break;

      case "comedy":
        thisAmount = 30000;

        if (perf.audience > 20) {
          thisAmount += 10000 + 500 * (perf.audience - 20);
        }

        thisAmount += 300 * perf.audience;
        break;

      default:
        throw new Error(`unknown type: ${play.type}`);
    }

    // Добавление бонусов
    volumeCredits += Math.max(perf.audience - 30, 0);

    // Дополнительный бонус за каждые 10 комедий
    if ("comedy" === play.type) volumeCredits += Math.floor(
          perf.audience / 5);

    // Вывод строки счета
    result += `  ${play.name}: ${format(thisAmount / 100)}`;
    result += ` (${perf.audience} seats)\n`;
    totalAmount += thisAmount;
  }
```

```
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
}
```

Выполнение этого кода для тестовых файлов данных, показанных выше, приводит к следующему результату:

```
Statement for BigCo
  Hamlet: $650.00 (55 seats)
  As You Like It: $580.00 (35 seats)
  Othello: $500.00 (40 seats)
Amount owed is $1,730.00
You earned 47 credits
```

Комментарии к программе

Что вы думаете о проекте этой программы? Первое, что я бы сказал о ней — это то, что она вполне терпима — такая короткая программа не требует какой-либо глубокой структуры, чтобы оставаться понятной. Но вспомните мое предыдущее замечание, что я должен использовать только маленькие примеры. Представьте себе эту программу в большем масштабе — возможно, из сотни строк. При таком размере будет трудно понять одну встроенную функцию.

Учитывая, что программа корректно работает, не является ли любое утверждение о ее структуре просто эстетическим суждением, вызванным неприязнью к "уродливому" коду? В конце концов, компилятору не важно, уродлив код или эстетически прекрасен. Но когда я вношу изменения, в процесс вовлекается человек, а людям не все равно. Плохо спроектированную систему трудно изменять, потому что трудно понять, что именно нужно изменить и как эти изменения будут взаимодействовать с существующим кодом, чтобы получить желаемое поведение. А если трудно понять, что и как изменять, то существует большая вероятность, что я буду допускать ошибки.

Таким образом, сталкиваясь с необходимостью модификации программы с сотнями строк кода, я бы предпочел, чтобы она была структурирована в виде набора функций и других элементов программы, которые облегчают понимание того, что делает программа. Если в программе отсутствует структура, мне обычно проще сначала добавить структуру в программу, а уж затем вносить необходимые изменения.

Если вам необходимо добавить в программу функциональную возможность, но код программы не структурирован способом, удобным для внесения изменений, — сначала выполните рефакторинг программы, чтобы упростить добавление функциональной возможности, и только затем приступайте к добавлению.

В нашем случае есть изменения, которые требуют пользователи. Во-первых, им нужен вывод в формате HTML. Давайте подумаем, на что повлияет такое изменение. Требуется добавление условных инструкций вокруг каждой инструкции добавления строку к результату. Это существенно усложнит функцию. Столкнувшись с таким требованием, большинство программистов предпочтут скопировать метод и изменить копию так, чтобы она выдавала HTML. Создание копии может показаться не слишком обременительной задачей, но все это — закладка разнообразных проблем на будущее. Любые изменения в логике оплаты заставили бы меня обновлять оба метода, причем обновлять согласованно. Если я пишу программу, которая никогда не будет изменяться, метод копирования и вставки может быть подходящим. Но если это долгоживущая программа, то дублирование представляет угрозу.

Это подводит меня ко второму изменению. Актеры хотят играть больше разнообразных пьес: они надеются добавить исторические, пасторальные, пасторально-комические, историко-пасторальные, трагически-исторические, трагически-комично-историко-пасторальные и прочие виды пьес в свой репертуар... И они еще точно не решили, что и когда хотят менять. Такое изменение повлияет как на оплату их работы, так и на способ расчета объема бонусов. Как опытный разработчик, я уверен, что, какую бы схему ни придумал заказчик, — не пройдет и полгода, как он снова изменит ее. Запросы на изменение программы приходят не как отдельные шпионы, а как маршевые батальоны.

Изменения классификации и оплаты должны вноситься в метод `statement`. Но если я копирую `statement` в `htmlStatement`, то мне нужно гарантировать согласованность любых изменений. Кроме того, по мере усложнения правил становится все труднее выяснять, куда именно вносить изменения, и труднее вносить их, не совершая ошибок.

Позвольте подчеркнуть, что именно эти изменения определяют необходимость проведения рефакторинга. Если код работает и его не нужно менять, то вполне можно оставить его в покое. Было бы неплохо улучшить его, но если кому-то не нужно его понимать, то он не причинит никакого реального вреда. Тем не менее, как только кто-то попытается разобраться в работе кода — с этим нужно что-то делать.

Первый шаг

При любом рефакторинге первый мой шаг всегда один и тот же. Я должен убедиться в наличии надежного набора тестов для той части кода, с которой буду работать. Тесты очень важны, поскольку даже при структурированном рефакторинге, который уменьшает возможность внесения ошибок, я остаюсь человеком, а человеку свойственно ошибаться. Чем больше программа — тем больше шансов, что что-то пойдет не так, поэтому мне нужны надежные тесты. В цифровую эпоху второе имя слабости — программное обеспечение

Поскольку метод `statement` возвращает строку, я создаю несколько счетов, вношу в каждый несколько пьес различного вида и генерирую соответствующие строки. Затем я сравниваю получаемую строку с образцовой, полученной вручную. Я настраиваю все эти тесты с помощью каркаса тестирования, чтобы иметь возможность запускать их с помощью простого нажатия клавиши в моей интегрированной среде разработки. Тесты занимают всего несколько секунд, и, как вы увидите, я запускаю их довольно часто.

Важной частью тестов является способ вывода результатов. Они выводят либо зеленый текст, свидетельствующий о совпадении с контрольными строками всех вычисляемых строк, либо красный список ошибок, т.е. строк, отличающихся от контрольных. Таким образом, тесты сами проверяют свои результаты. Это, опять же, очень важно, потому что иначе нужно тратить много времени на то, чтобы вручную сравнивать получаемые результаты с записанными на бумаге. Современные каркасы тестирования предоставляют все возможности для написания и выполнения самопроверяющихся тестов.

Перед тем как приступать к рефакторингу, убедитесь, что у вас есть комплект надежных самопроверяющихся тестов.

Выполняя рефакторинг, мы должны опираться на тесты. Я рассматриваю их как детектор, защищающий меня от моих собственных ошибок. Написав то, что я хочу, дважды — в коде и в тесте — я должен согласованно ошибиться в обоих местах, чтобы обмануть детектор. Дважды проверяя свою работу, я уменьшаю шансы сделать что-то не так. Хотя для создания тестов требуется время, в конечном итоге я оказываюсь со значительным выигрышем, тратя куда меньше времени на отладку. Это очень важная часть рефакторинга, которой посвящена целая глава 4, "Создание тестов".

Декомпозиция функции `statement`

При рефакторинге длинной функции, подобной этой, я мысленно пытаюсь определить точки, которые разделяют различные части общего поведения. Первый кусок, который бросается в глаза, — это оператор `switch` в средине функции.

```
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;

  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                       { style: "currency", currency: "USD",
                         minimumFractionDigits: 2 }).format;

  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = 0;

    switch (play.type) {
      case "tragedy":
        thisAmount = 40000;
        if (perf.audience > 30) {
          thisAmount += 1000 * (perf.audience - 30);
        }
        break;
      case "comedy":
        thisAmount = 30000;
        if (perf.audience > 20) {
          thisAmount += 10000 + 500 * (perf.audience - 20);
        }
        thisAmount += 300 * perf.audience;
        break;
      default:
        throw new Error(`unknown type: ${play.type}`);
    }

    // Добавление бонусов
    volumeCredits += Math.max(perf.audience - 30, 0);

    // Дополнительный бонус за каждые 10 комедий
    if ("comedy" === play.type) volumeCredits += Math.floor(
            perf.audience / 5);

    // Вывод строки счета
    result += `  ${play.name}: ${format(thisAmount / 100)}`;
    result += ` (${perf.audience} seats)\n`;
    totalAmount += thisAmount;
  }
```

```
  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
}
```

Рассматривая этот блок, я делаю вывод, что он рассчитывает плату за одно исполнение. И этот вывод является частью понимания кода. Но, как выразился Уорд Каннингем (Ward Cunningham), это понимание у меня в голове, которая представляет собой общеизвестно изменчивую форму хранения. Мне нужно сохранить это знание, переместив его из головы обратно в код, чтобы, если я вернусь к нему позже, код сам сказал мне, что он делает, и мне не пришлось бы снова это выяснять.

Способ внедрить это понимание в код состоит в том, чтобы превратить данный фрагмент кода в отдельную функцию, дав ей название, ясно говорящее о том, что она делает, — что-то вроде `amountFor(aPerformance)`. Чтобы превратить фрагмент кода в функцию, я использую процедуру, которая минимизирует возможность ошибки, — *Извлечение функции* (с. 152).

Во-первых, мне нужно выявить во фрагменте любые переменные, которые больше не будут находиться в области видимости после того, как я извлеку код в отдельную функцию. В данном случае у меня их три: `perf`, `play` и `thisAmount`. Первые две используются извлекаемым кодом, но не модифицируются, поэтому я могу передать их в качестве параметров. Модифицируемые переменные нуждаются в большем внимании. Здесь есть только одна такая переменная, поэтому я могу просто вернуть ее из функции. Я также могу выполнить ее инициализацию внутри извлеченного кода. Вот что у меня в результате получается.

```
function statement...

function amountFor(perf, play) {
  let thisAmount = 0;

  switch (play.type) {
    case "tragedy":
      thisAmount = 40000;

      if (perf.audience > 30) {
        thisAmount += 1000 * (perf.audience - 30);
      }

      break;

    case "comedy":
      thisAmount = 30000;

      if (perf.audience > 20) {
        thisAmount += 10000 + 500 * (perf.audience - 20);
      }
```

```
      thisAmount += 300 * perf.audience;
      break;

    default:
      throw new Error(`unknown type: ${play.type}`);
  }

  return thisAmount;
}
```

Заголовок вида *function someName...* перед кодом означает, что этот код находится в области видимости функции, файла или класса, указанного в заголовке. Обычно внутри этой области есть и другой код, который я не показываю, так как сейчас я его не обсуждаю.

Исходный код `statement` теперь вызывает эту функцию для заполнения `thisAmount`:

Верхний уровень...

```
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;

  let result = `Statement for ${invoice.customer}\n`;
const format = new Intl.NumberFormat("en-US",
                    { style: "currency", currency: "USD",
                      minimumFractionDigits: 2}).format;

  for (let perf of invoice.performances) {
    const play = plays[perf.playID];
    let thisAmount = amountFor(perf, play);

    // Добавление бонусов
    volumeCredits += Math.max(perf.audience - 30, 0);

    // Дополнительный бонус за каждые 10 комедий
    if ("comedy" === play.type) volumeCredits += Math.floor(
            perf.audience / 5);

    // Вывод строки счета
    result += `  ${play.name}: ${format(thisAmount / 100)}`;
    result += ` (${perf.audience} seats)\n`;
    totalAmount += thisAmount;
  }

  result += `Amount owed is ${format(totalAmount/100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

Внеся это изменение, я немедленно выполнил компиляцию и проверил, не сломал ли я что-нибудь. Это важная привычка — выполнять проверки после каждого рефакторинга, пусть и простого. Ошибки легко сделать — по крайней мере я делаю их легко. Тестирование после каждого изменения означает, что когда, я делаю ошибку, у меня внесено только очень небольшое изменение, что значительно облегчает поиск ошибки и ее исправление. В этом суть процесса рефакторинга: небольшие изменения и немедленное тестирование. Если я попытаюсь сделать сразу слишком много, то из-за ошибки мне придется заниматься сложной отладкой, которая может занять много времени. Небольшие изменения, обеспечивающие быструю обратную связь, являются ключом к предотвращению этой неприятности.

Я использую термин *компиляция* для обозначения всего, что нужно, чтобы создать исполняемый файл JavaScript. Поскольку JavaScript исполняется непосредственно интерпретатором языка, это может не означать ничего, но в других случаях это может означать перемещение кода в выходной каталог и/или применение процессора, такого как Babel[1].

В процессе рефакторинга изменяйте программу небольшими порциями, это облегчает обнаружение ошибок.

Так как это JavaScript, я могу извлечь `amountFor` во вложенную функцию `statement`. Это полезно, так как означает, что мне не нужно передавать данные, которые находятся внутри области видимости функции, во вновь извлеченную функцию. В данном случае это не имеет значения, но на одну проблему, с которой нужно иметь дело, становится меньше.

Итак, тесты пройдены, поэтому мой следующий шаг — зафиксировать изменения в локальной системе контроля версий. Я использую систему контроля версий, которая позволяет делать частные фиксации кода (такую как *git* или *mercurial*). Я выполняю фиксацию после каждого успешного рефакторинга, поэтому могу легко вернуться к последнему рабочему состоянию, если позже что-либо испорчу. Затем я объединяю изменения в более значительные фиксации, прежде чем помещать изменения в общее совместно используемое хранилище кода.

Извлечение функции (с. 152) — это распространенный автоматизируемый рефакторинг. Если бы я программировал на Java, то инстинктивно нажал бы соответствующую последовательность клавиш в моей интегрированной среде разработки, чтобы выполнить этот рефакторинг. Пока что я пишу все это вручную, поскольку надежной поддержки рефакторинга в инструментах JavaScript нет. Это не сложно, хотя следует быть осторожным с локальными переменными.

[1] `https://babeljs.io`

Использовав рефакторинг *Извлечение функции* (с. 152), я просматриваю результат работы, чтобы увидеть, нет ли каких-нибудь быстрых и простых изменений, которые я могу внести, чтобы прояснить извлеченную функцию. Первое, что я делаю, — это переименовываю некоторые переменные, чтобы сделать их более понятными, например изменяю `thisAmount` на `result`.

```
function statement...
function amountFor(perf, play) {
  let result = 0;
  switch (play.type) {
    case "tragedy":
      result = 40000;
      if (perf.audience > 30) {
        result += 1000 * (perf.audience - 30);
      }
      break;
    case "comedy":
      result = 30000;
      if (perf.audience > 20) {
        result += 10000 + 500 * (perf.audience - 20);
      }
      result += 300 * perf.audience;
      break;
    default:
      throw new Error(`unknown type: ${play.type}`);
  }
  return result;
}
```

Это мой собственный стандарт стиля кодирования — всегда называть возвращаемое из функции значение `result`. Таким образом, я всегда знаю его роль. Я снова компилирую, тестирую и фиксирую. Затем я перехожу к первому аргументу.

```
function statement...
function amountFor(aPerformance, play) {
  let result = 0;
  switch (play.type) {
    case "tragedy":
      result = 40000;
      if (aPerformance.audience > 30) {
        result += 1000 * (aPerformance.audience - 30);
      }
      break;
    case "comedy":
      result = 30000;
      if (aPerformance.audience > 20) {
        result += 10000 + 500 * (aPerformance.audience - 20);
      }
```

```
      result += 300 * aPerformance.audience;
      break;
    default:
      throw new Error(`unknown type: ${play.type}`);
  }
  return result;
}
```

Этот код вновь соответствует моему стилю. В динамически типизированном языке, таком как JavaScript, полезно отслеживать типы, поэтому имя параметра по умолчанию включает в себя имя типа. Я использую неопределенный артикль в имени, если в нем нет какой-то конкретной информации о роли. Я узнал об этом соглашении от Кента Бека [4] и считаю его весьма полезным.

Любой в состоянии написать код, который будет понятен компилятору. Хороший программист пишет код, который будет понятен человеку.

Стоило ли это переименование затраченных усилий? Несомненно. Хороший код должен четко сообщать, что он делает, а имена переменных являются ключом к хорошему коду. Никогда не бойтесь менять имена, чтобы сделать их понятными. С наличием хороших инструментов поиска и замены это обычно не сложно; тестирование и статическая типизация в языке, который ее поддерживает, обнаружат все пропущенные имена. А с помощью автоматизированных инструментов рефакторинга легко переименовать даже широко используемые функции.

Далее следовало бы переименовать параметр `play`, но на него у меня другие планы.

Устранение переменной `play`

Поскольку я рассматриваю параметры для `amountFor`, я смотрю, откуда они берутся. `aPerformance` является переменной цикла, поэтому, естественно, она меняется с каждой итерацией цикла. Но `play` вычисляется из `aPerformance`, поэтому нет необходимости передавать ее в качестве параметра — ее можно просто пересчитать внутри `amountFor`. Когда я разбиваю длинную функцию, мне нравится избавляться от таких переменных, как `play`, потому что временные переменные создают много локальных имен, усложняющих процесс извлечения. Рефакторинг, который я буду использовать здесь, — *Замена временной переменной запросом* (с. 225).

Я начинаю с выделения правой части присваивания в функцию.

```
function statement...
function playFor(aPerformance) {
    return plays[aPerformance.playID];
}
```

```
Верхний уровень...
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;

  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                       { style: "currency", currency: "USD",
                         minimumFractionDigits: 2 }).format;

  for (let perf of invoice.performances) {
    const play = playFor(perf);
    let thisAmount = amountFor(perf, play);
    // Добавление бонусов
    volumeCredits += Math.max(perf.audience - 30, 0);

    // Дополнительный бонус за каждые 10 комедий
    if ("comedy" === play.type) volumeCredits += Math.floor(
            perf.audience / 5);

    // Вывод строки счета
    result += `  ${play.name}: ${format(thisAmount / 100)}`;
    result += ` (${perf.audience} seats)\n`;
    totalAmount += thisAmount;
  }

  result += `Amount owed is ${format(totalAmount / 100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

Выполняю компиляцию-тестирование-фиксацию и применяю рефакторинг *Встраивание переменной* (с. 169).

```
Верхний уровень...
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;

  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                       { style: "currency", currency: "USD",
                         minimumFractionDigits: 2 }).format;

  for (let perf of invoice.performances) {
    const play = playFor(perf);
    let thisAmount = amountFor(perf, playFor(perf));
    // Добавление бонусов
    volumeCredits += Math.max(perf.audience - 30, 0);

    // Дополнительный бонус за каждые 10 комедий
    if ("comedy" === playFor(perf).type)
        volumeCredits += Math.floor(perf.audience / 5);
```

```
    // Вывод строки счета
    result += `  ${playFor(perf).name}: ${format(thisAmount/100)}`;
    result += ` (${perf.audience} seats)\n`;
    totalAmount += thisAmount;
  }

  result += `Amount owed is ${format(totalAmount / 100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

Провожу компиляцию-тестирование-фиксацию. После этого можно применить рефакторинг *Изменение объявления функции* (с. 170) к `amountFor` для удаления параметра `play`. Я делаю это в два этапа. Сначала использую новую функцию внутри `amountFor`.

```
function statement...
function amountFor(aPerformance, play) {
  let result = 0;
  switch (playFor(aPerformance).type) {
    case "tragedy":
      result = 40000;
      if (aPerformance.audience > 30) {
        result += 1000 * (aPerformance.audience - 30);
      }
      break;
    case "comedy":
      result = 30000;
      if (aPerformance.audience > 20) {
        result += 10000 + 500 * (aPerformance.audience - 20);
      }
      result += 300 * aPerformance.audience;
      break;
    default:
      throw new Error(`unknown type: ${playFor(aPerformance).type}`);
  }
  return result;
}
```

Выполняю компиляцию-тестирование-фиксацию, а затем удаляю параметр.

```
Верхний уровень...
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;

  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                       { style: "currency", currency: "USD",
                         minimumFractionDigits: 2}).format;
```

```
  for (let perf of invoice.performances) {
    let thisAmount = amountFor(perf~~, playFor(perf)~~);

    // Добавление бонусов
    volumeCredits += Math.max(perf.audience - 30, 0);

    // Дополнительный бонус за каждые 10 комедий
    if ("comedy" === playFor(perf).type)
        volumeCredits += Math.floor(perf.audience / 5);

    // Вывод строки счета
    result += `  ${playFor(perf).name}: ${format(thisAmount/100)}`;
    result += ` (${perf.audience} seats)\n`;
    totalAmount += thisAmount;
  }

  result += `Amount owed is ${format(totalAmount / 100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
function statement...
function amountFor(aPerformance~~, play~~) {
  let result = 0;
  switch (playFor(aPerformance).type) {
    case "tragedy":
      result = 40000;
      if (aPerformance.audience > 30) {
        result += 1000 * (aPerformance.audience - 30);
      }
      break;
    case "comedy":
      result = 30000;
      if (aPerformance.audience > 20) {
        result += 10000 + 500 * (aPerformance.audience - 20);
      }
      result += 300 * aPerformance.audience;
      break;
    default:
      throw new Error(`unknown type: ${playFor(aPerformance).type}`);
  }
  return result;
}
```

И вновь я выполняю компиляцию-тестирование-фиксацию.

Этот рефакторинг может тревожить некоторых программистов. Ранее код поиска выполнялся в каждой итерации цикла один раз; сейчас он исполняется трижды. Я еще поговорю о взаимодействии рефакторинга и производительности позже, а сейчас просто замечу, что такое изменение вряд ли существенно повлияет на производительность. Но даже если бы это было так, то гораздо проще повысить производительность хорошо факторизованного кода.

Большим преимуществом удаления локальных переменных является то, что это делает процесс извлечения намного проще, поскольку приходится работать с меньшей областью видимости. Обычно я убираю локальные переменные, прежде чем делать какие-либо извлечения.

Закончив работу с аргументами `amountFor`, я оглядываюсь на место вызова этой функции. Она используется для установки временной переменной, которая больше не обновляется, поэтому я применяю рефакторинг *Встраивание переменной* (с. 169).

```
Верхний уровень...
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;

  let result = `Statement for ${invoice.customer}\n`;
const format = new Intl.NumberFormat("en-US",
                    { style: "currency", currency: "USD",
                      minimumFractionDigits: 2}).format;

  for (let perf of invoice.performances) {
    // Добавление бонусов
    volumeCredits += Math.max(perf.audience - 30, 0);

    // Дополнительный бонус за каждые 10 комедий
    if ("comedy" === playFor(perf).type)
        volumeCredits += Math.floor(perf.audience / 5);

    // Вывод строки счета
    result += `  ${playFor(perf).name}: `;
    result += `${format(amountFor(perf)/100)} `;
    result += `(${perf.audience} seats)\n`;
    totalAmount += amountFor(perf);
  }

  result += `Amount owed is ${format(totalAmount / 100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

Извлечение бонусов

Итак, вот текущее состояние тела функции `statement`:

```
Верхний уровень...
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
```

```
  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                       { style: "currency", currency: "USD",
                         minimumFractionDigits: 2}).format;

  for (let perf of invoice.performances) {
    // Добавление бонусов
    volumeCredits += Math.max(perf.audience - 30, 0);

    // Дополнительный бонус за каждые 10 комедий
    if ("comedy" === playFor(perf).type)
        volumeCredits += Math.floor(perf.audience / 5);

    // Вывод строки счета
    result += `  ${playFor(perf).name}: `;
    result += `${format(amountFor(perf)/100)} `;
    result += `(${perf.audience} seats)\n`;
    totalAmount += amountFor(perf);
  }

  result += `Amount owed is ${format(totalAmount / 100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

Теперь я получаю выгоду от удаления переменной `play`, поскольку оно упрощает извлечение подсчета бонусов путем удаления одной из локальных переменных. Но я все еще вынужден иметь дело с двумя другими переменными. Переменную `perf` передать легко, но с `volumeCredits` ситуация немного сложнее, так как это аккумулятор, обновляемый на каждом проходе цикла. Поэтому лучше всего инициализировать ее тень внутри извлеченной функции и возвращать ее.

```
function statement...
function volumeCreditsFor(perf) {
  let volumeCredits = 0;
  volumeCredits += Math.max(perf.audience - 30, 0);

  if ("comedy" === playFor(perf).type)
    volumeCredits += Math.floor(perf.audience / 5);

  return volumeCredits;
}

Верхний уровень...
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;

  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                       { style: "currency", currency: "USD",
                         minimumFractionDigits: 2}).format;
```

```
  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);

    // Вывод строки счета
    result += `  ${playFor(perf).name}: `;
    result += `${format(amountFor(perf)/100)} `;
    result += `(${perf.audience} seats)\n`;
    totalAmount += amountFor(perf);
  }
  result += `Amount owed is ${format(totalAmount / 100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

Я удаляю ненужный (и в данном случае вводящий в заблуждение) комментарий. Затем выполняю компиляцию-тестирование-фиксацию и переименовываю переменные в новой функции.

```
function statement...
function volumeCreditsFor(aPerformance) {
  let result = 0;
  result += Math.max(aPerformance.audience - 30, 0);

  if ("comedy" === playFor(aPerformance).type)
    result += Math.floor(aPerformance.audience / 5);

  return result;
}
```

Здесь все показано как выполненное за один шаг, но я проводил переименования по одному, с компиляцией-тестированием-фиксацией после каждого переименования.

Удаление переменной `format`

Давайте еще раз рассмотрим функцию `statement`:

```
Верхний уровень...
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;

  let result = `Statement for ${invoice.customer}\n`;
  const format = new Intl.NumberFormat("en-US",
                    { style: "currency", currency: "USD",
                      minimumFractionDigits: 2 }).format;

  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);

    // Вывод строки счета
    result += `  ${playFor(perf).name}: `;
```

```
    result += `${format(amountFor(perf)/100)} `;
    result += `(${perf.audience} seats)\n`;
    totalAmount += amountFor(perf);
  }

  result += `Amount owed is ${format(totalAmount / 100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

Как я уже говорил ранее, временные переменные могут оказаться проблемой. Они полезны только в пределах их собственных подпрограмм и ведут к длинным, сложным подпрограммам. Мой следующий шаг — заменить некоторые из них. Простейшим случаем является переменная `format` — это просто присвоение функции временной переменной, которое я предпочитаю заменить объявленной функцией.

```
function statement...
function format(aNumber) {
  return new Intl.NumberFormat("en-US",
                  { style: "currency", currency: "USD",
                    minimumFractionDigits: 2 }).format(aNumber);
}

Верхний уровень...
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;

  let result = `Statement for ${invoice.customer}\n`;

  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);
    // Вывод строки счета
    result += `  ${playFor(perf).name}: `;
    result += `${format(amountFor(perf)/100)}`;
    result += `(${perf.audience} seats)\n`;
    totalAmount += amountFor(perf);
    totalAmount += amountFor(perf);
  }

  result += `Amount owed is ${format(totalAmount / 100)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

Хотя замена переменной функции на объявленную функцию является рефакторингом, я не называл ее и не включил в каталог. Существует много рефакторингов, которые я не чувствовал достаточно важными для этого. Данный рефакторинг простой и относительно редкий, поэтому я не думаю, что он стоит отдельного имени и строки в каталоге.

Меня также не устраивает название — оно недостаточно точно передает то, что делает функция. Название `formatAsUSD` будет слишком длинным, поскольку оно используется в шаблоне строки, особенно такой небольшой области видимости. Я думаю, что следует подчеркнуть тот факт, что это форматирование денежной суммы, а потому я выбираю имя, которое указывает именно это, и применяю рефакторинг *Изменение объявления функции* (с. 170).

```
Верхний уровень...
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;

  let result = `Statement for ${invoice.customer}\n`;

  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);
    // Вывод строки счета
    result += ` ${playFor(perf).name}: `;
    result += `${usd(amountFor(perf))} (${perf.audience} seats)\n`;
    totalAmount += amountFor(perf);
  }

  result += `Amount owed is ${usd(totalAmount)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;

function statement...
function usd(aNumber) {
  return new Intl.NumberFormat("en-US",
               { style: "currency", currency: "USD",
                 minimumFractionDigits: 2}).format(aNumber/100);
  }
```

Именовать одновременно и важно, и сложно. Разбиение большой функции на более мелкие только улучшает код, если имена хороши. При наличии хороших имен мне не нужно обращаться к телу функции, чтобы понять, что она делает. Но с первого раза дать имена правильно весьма трудно, поэтому я использую наилучшее имя, какое только могу себе представить в данный момент, и без колебаний выполняю переименования позже. Зачастую для того, чтобы понять, какое имя действительно является наилучшим, требуется дополнительный проход по коду.

При изменении имени я заодно перемещаю дублированное деление на 100 в функцию. Хранение денег в виде целого количества центов является распространенным подходом, которое позволяет избежать опасностей хранения дробных денежных значений в виде чисел с плавающей запятой, но позволяет мне использовать арифметические операторы. Однако всякий раз, когда я хочу отобразить такое значение, мне нужно десятичное число; поэтому моя функция форматирования должна позаботиться о делении.

Удаление переменной volumeCredits

Моя следующая цель — переменная volumeCredits. Это более сложный случай, так как она используется во время итераций цикла. Мой первый шаг состоит в использовании рефакторинга *Разделение цикла* (с. 274) для выделения накопления volumeCredits.

```
Верхний уровень...
function statement(invoice, plays) {
  let totalAmount = 0;
  let volumeCredits = 0;
  let result = `Statement for ${invoice.customer}\n`;

  for (let perf of invoice.performances) {
    // Вывод строки счета
    result += ` ${playFor(perf).name}: `;
    result += `${usd(amountFor(perf))} (${perf.audience} seats)\n`;
    totalAmount += amountFor(perf);
  }

  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);
  }

  result += `Amount owed is ${usd(totalAmount)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

После этого я могу использовать рефакторинг *Перемещение инструкций* (с. 269), чтобы переместить объявление переменной за цикл.

```
Верхний уровень...
function statement(invoice, plays) {
  let totalAmount = 0;
  let result = `Statement for ${invoice.customer}\n`;

  for (let perf of invoice.performances) {
    // Вывод строки счета
    result += ` ${playFor(perf).name}: `;
    result += `${usd(amountFor(perf))} (${perf.audience} seats)\n`;
    totalAmount += amountFor(perf);
  }

  let volumeCredits = 0;

  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);
  }

  result += `Amount owed is ${usd(totalAmount)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

Сбор вместе всего кода, который обновляет переменную volumeCredits, облегчает выполнение рефакторинга *Замена временной переменной запросом* (с. 225). Как и ранее, первым шагом является применение рефакторинга *Извлечение функции* (с. 152) к общему вычислению переменной.

```
function statement...
function totalVolumeCredits() {
  let volumeCredits = 0;

  for (let perf of invoice.performances) {
    volumeCredits += volumeCreditsFor(perf);
  }

  return volumeCredits;
}

Верхний уровень...
function statement(invoice, plays) {
  let totalAmount = 0;

  let result = `Statement for ${invoice.customer}\n`;

  for (let perf of invoice.performances) {
    // Вывод строки счета
    result += ` ${playFor(perf).name}: `;
    result += `${usd(amountFor(perf))} (${perf.audience} seats)\n`;
    totalAmount += amountFor(perf);
  }

  let volumeCredits = totalVolumeCredits();
  result += `Amount owed is ${usd(totalAmount)}\n`;
  result += `You earned ${volumeCredits} credits\n`;
  return result;
```

После извлечения я могу применить рефакторинг *Встраивание переменной* (с. 169):

```
Верхний уровень...
function statement(invoice, plays) {
  let totalAmount = 0;

  let result = `Statement for ${invoice.customer}\n`;

  for (let perf of invoice.performances) {
    // Вывод строки счета
    result += ` ${playFor(perf).name}: `;
    result += `${usd(amountFor(perf))} (${perf.audience} seats)\n`;
    totalAmount += amountFor(perf);
  }
```

```
result += `Amount owed is ${usd(totalAmount)}\n`;
result += `You earned ${totalVolumeCredits()} credits\n`;
return result;
```

Позвольте мне немного остановиться, чтобы поговорить о том, что я только что сделал. Во-первых, я знаю, что в связи с этим изменением многие снова будут беспокоиться о производительности, так как люди приучены опасаться повторений кода в цикле. Но в большинстве случаев работа такого цикла влияет на производительность незначительно. Если бы вы замерили время работы до и после этого рефакторинга, то, вероятно, не заметили бы существенного изменения в скорости. Большинство программистов, даже опытных, плохо разбираются в том, как на самом деле работает код. При использовании умных компиляторов, современных методов кеширования и тому подобного многие из наших интуитивных представлений оказываются ложными. Производительность программного обеспечения обычно зависит только от небольшого количества фрагментов кода, и внесение изменений в других местах не имеет заметного значения.

Но "в большинстве случаев" — не то же самое, что и "всегда". Иногда рефакторинг может существенно повлиять на производительность. Но даже тогда я обычно не останавливаюсь и продолжаю работу, потому что намного проще повысить производительность хорошо продуманного кода. Если во время рефакторинга я обнаруживаю существенную проблему с производительностью — впоследствии я занимаюсь ее решением. Может случиться и так, что это ведет меня к отмене некоторого сделанного ранее рефакторинга, но чаще всего благодаря рефакторингу я могу применить более эффективный способ повышения производительности. В конечном итоге я получаю код, который оказывается и понятнее, и быстрее.

Итак, мой глобальный совет по вопросам производительности при рефакторинге: в большинстве случаев вы должны ее игнорировать. Если ваш рефакторинг приводит к снижению производительности, сначала завершите его, а затем разберитесь с производительностью.

Второй аспект, на который я хочу обратить ваше внимание, заключается в том, насколько малы были шаги при удалении `volumeCredits`. Вот четыре рефакторинга, каждый из которых сопровождался компиляцией, тестированием и фиксацией в моем локальном хранилище исходного кода.

- *Разделение цикла* (с. 274) для выделения накопления.
- *Перемещение инструкций* (с. 269) для переноса инициализирующего кода к коду накопления.
- *Извлечение функции* (с. 152 для создания функции для расчета суммы.
- *Встраивание переменной* (с. 169) для полного удаления переменной.

Признаюсь, я не всегда делаю такие короткие шаги, но всякий раз, когда что-то становится слишком сложным, моя первая реакция — делать шаги покороче.

В частности, если во время рефакторинга оказывается не пройденным некоторый тест, то, если я не смогу сразу увидеть и исправить проблему, я вернусь к последней корректно работавшей фиксации изменений кода и выполню те же действия с помощью еще меньших шагов. Это работает, потому что я выполняю фиксации кода очень часто, и потому что ключом к быстрому продвижению являются маленькие шаги, особенно при работе со сложным кодом.

Затем я повторяю описанную последовательность, чтобы удалить `totalAmount`. Я начинаю с разделения цикла (компиляция-тестирование-фиксация), затем перемещаю инициализацию переменной (компиляция-тестирование-фиксация) и извлекаю функцию. Здесь есть одно замечание: наилучшее имя для функции — `totalAmount`, но это имя переменной, и я не могу использовать оба имени одновременно. Поэтому я даю новой функции при извлечении случайное имя (и не забываю о компиляции-тестировании-фиксации).

```
function statement...
function appleSauce() {
  let totalAmount = 0;

  for (let perf of invoice.performances) {
    totalAmount += amountFor(perf);
  }

  return totalAmount;
}

Верхний уровень...
function statement(invoice, plays) {

  let result = `Statement for ${invoice.customer}\n`;

  for (let perf of invoice.performances) {
    result += ` ${playFor(perf).name}: `;
    result += `${usd(amountFor(perf))} (${perf.audience} seats)\n`;
  }

  let totalAmount = appleSauce();
  result += `Amount owed is ${usd(totalAmount)}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;
```

Затем я встраиваю переменную (компиляция-тестирование-фиксация) и переименовываю функцию, давая ей более разумное имя (компиляция-тестирование-фиксация).

```
Верхний уровень...
function statement(invoice, plays) {

  let result = `Statement for ${invoice.customer}\n`;
```

```
  for (let perf of invoice.performances) {
    result += ` ${playFor(perf).name}: `;
    result += `${usd(amountFor(perf))} (${perf.audience} seats)\n`;
  }

  result += `Amount owed is ${usd(totalAmount())}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;

function statement...
function totalAmount() {
  let totalAmount = 0;
  for (let perf of invoice.performances) {
    totalAmount += amountFor(perf);
  }
  return totalAmount;
}
```

Я также изменяю имена переменных внутри извлеченных функций, придерживаясь своего описанного ранее соглашения об имени `result`.

```
function statement...
function totalAmount() {
  let result = 0;
  for (let perf of invoice.performances) {
    result += amountFor(perf);
  }
  return result;
}

function totalVolumeCredits() {
  let result = 0;
  for (let perf of invoice.performances) {
    result += volumeCreditsFor(perf);
  }
  return result;
}
```

Состояние: множество вложенных функций

Сейчас самое время сделать паузу и взглянуть на общее состояние кода:

```
function statement(invoice, plays) {
  let result = `Statement for ${invoice.customer}\n`;
  for (let perf of invoice.performances) {
    result += ` ${playFor(perf).name}: `;
    result += `${usd(amountFor(perf))}(${perf.audience} seats)\n`;
  }
  result += `Amount owed is ${usd(totalAmount())}\n`;
```

```
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;

  function totalAmount() {
    let result = 0;
    for (let perf of invoice.performances) {
      result += amountFor(perf);
    }
    return result;
  }

  function totalVolumeCredits() {
    let result = 0;
    for (let perf of invoice.performances) {
      result += volumeCreditsFor(perf);
    }
    return result;
  }

  function usd(aNumber) {
    return new Intl.NumberFormat("en-US",
                { style: "currency", currency: "USD",
                  minimumFractionDigits: 2}).format(aNumber / 100);
  }

  function volumeCreditsFor(aPerformance) {
    let result = 0;
    result += Math.max(aPerformance.audience - 30, 0);
    if ("comedy" === playFor(aPerformance).type)
        result += Math.floor(aPerformance.audience / 5);
    return result;
  }

  function playFor(aPerformance) {
    return plays[aPerformance.playID];
  }

  function amountFor(aPerformance) {
    let result = 0;
    switch (playFor(aPerformance).type) {
      case "tragedy":
        result = 40000;
        if (aPerformance.audience > 30) {
          result += 1000 * (aPerformance.audience - 30);
        }
        break;
      case "comedy":
        result = 30000;
        if (aPerformance.audience > 20) {
          result += 10000 + 500 * (aPerformance.audience - 20);
        }
```

```
        result += 300 * aPerformance.audience;
        break;
      default:
        throw
          new Error(`unknown type: ${playFor(aPerformance).type}`);
    }

    return result;
  }
}
```

Структура кода теперь намного лучше, чем была ранее. Функция верхнего уровня `statement` теперь представляет собой всего восемь строк кода, и все, что она делает, — это форматирует вывод. Вся логика вычислений перенесена в несколько вспомогательных функций. Это облегчает понимание каждого отдельного расчета, а также общего потока выполнения отчета.

Разделение вычисления и форматирования

До сих пор мой рефакторинг был сосредоточен на улучшении структурности функции, чтобы ее можно было понять и увидеть с точки зрения разделения на логические части. Это часто бывает в самом начале рефакторинга. Важно разбить большие и сложные куски на мелкие кусочки, а также дать им правильные имена. Сосредоточимся на изменении функциональности, которое хочу внести; в частности — на разработке HTML-версии. Теперь это намного легче сделать во многих отношениях. Когда весь код вычислений отделен, мне достаточно написать HTML-версию из нескольких строк кода. Проблема в том, что эти выделенные функции вложены в метод получения текста, и я не хочу копировать и вставлять их в новую функцию, какой бы хорошо организованной она ни была. Я хочу, чтобы в текстовой и HTML-версиях использовались одинаковые функции вычислений.

Есть разные способы сделать это, но один из моих любимых методов — применение рефакторинга *Разделение этапа* (с. 201). Моя цель состоит в том, чтобы разделить логику на две части: одну, которая вычисляет необходимые данные, и другую, которая переводит их в текст или HTML. Первый этап создает промежуточную структуру данных, которая передается второму этапу.

Я начинаю рефакторинг *Разделение этапа* (с. 201), применяя рефакторинг *Извлечение функции* (с. 152) к коду, составляющему второй этап. В данном случае это код вывода, который фактически является всем содержимым функции `statement`. Вместе со всеми вложенными функциями он входит в функцию верхнего уровня, которую я назвал `renderPlainText`.

```
function statement(invoice, plays) {
  return renderPlainText(invoice, plays);
}

function renderPlainText(invoice, plays) {

  let result = `Statement for ${invoice.customer}\n`;

  for (let perf of invoice.performances) {
    result += `  ${playFor(perf).name}: `;
    result += `${usd(amountFor(perf))} (${perf.audience} seats)\n`;
  }

  result += `Amount owed is ${usd(totalAmount())}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;

  function totalAmount() { ... }
  function totalVolumeCredits() { ... }
  function usd(aNumber) { ... }
  function volumeCreditsFor(aPerformance) { ... }
  function playFor(aPerformance) { ... }
  function amountFor(aPerformance) { ... }
```

Я выполняю обычные компиляцию-тестирование-фиксацию, затем создаю объект, который будет действовать как промежуточная структура данных между двумя фазами. Затем передаю этот объект данных в качестве аргумента функции `renderPlainText` (компиляция-тестирование-фиксация).

```
function statement(invoice, plays) {
  const statementData = {};
  return renderPlainText(statementData, invoice, plays);
}

function renderPlainText(data, invoice, plays) {
  let result = `Statement for ${invoice.customer}\n`;
  for (let perf of invoice.performances) {
    result += `  ${playFor(perf).name}: `;
    result += `${usd(amountFor(perf))} (${perf.audience} seats)\n`;
  }
  result += `Amount owed is ${usd(totalAmount())}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;

  function totalAmount() {...}
  function totalVolumeCredits() {...}
  function usd(aNumber) {...}
  function volumeCreditsFor(aPerformance) {...}
  function playFor(aPerformance) {...}
  function amountFor(aPerformance) {...}
```

Теперь я рассматриваю другие аргументы, используемые функцией `renderPlainText`. Я хочу переместить поступающие от них данные в промежуточную структуру данных, с тем чтобы весь код вычислений перемести в функцию `statement`, а функция `renderPlainText` работала только с данными, передаваемыми ей через соответствующий параметр.

Мой первый шаг состоит в том, чтобы добавить клиент к промежуточному объекту (компиляция-тестирование-фиксация).

```
function statement (invoice, plays) {
  const statementData = {};
  statementData.customer = invoice.customer;
  return renderPlainText(statementData, invoice, plays);
}

function renderPlainText(data, invoice, plays) {
  let result = `Statement for ${data.customer}\n`;
  for (let perf of invoice.performances) {
    result += ` ${playFor(perf).name}: `;
    result += `${usd(amountFor(perf))} (${perf.audience} seats)\n`;
  }
  result += `Amount owed is ${usd(totalAmount())}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;
```

Точно так же я добавляю представления, что позволяет мне удалить параметр `invoice` из `renderPlainText` (компиляция-тестирование-фиксация).

```
Верхний уровень...
function statement (invoice, plays) {
  const statementData = {};
  statementData.customer = invoice.customer;
  statementData.performances = invoice.performances;
  return renderPlainText(statementData, ~~invoice,~~ plays);
}

function renderPlainText(data, plays) {
  let result = `Statement for ${data.customer}\n`;
  for (let perf of data.performances) {
    result += ` ${playFor(perf).name}: `;
    result += `${usd(amountFor(perf))} (${perf.audience} seats)\n`;
  }
  result += `Amount owed is ${usd(totalAmount())}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;

function renderPlainText...
function totalAmount() {
  let result = 0;
  for (let perf of data.performances) {
    result += amountFor(perf);
  }
```

```
  return result;
}

function totalVolumeCredits() {
  let result = 0;
  for (let perf of data.performances) {
    result += volumeCreditsFor(perf);
  }
  return result;
}
```

Теперь я хотел бы, чтобы название поступало из промежуточных данных. Для этого мне нужно добавить в запись соответствующие данные (компиляция-тестирование-фиксация).

```
function statement (invoice, plays) {
  const statementData = {};
  statementData.customer = invoice.customer;
  statementData.performances =
    invoice.performances.map(enrichPerformance);
  return renderPlainText(statementData, plays);

function enrichPerformance(aPerformance) {
  const result = Object.assign({}, aPerformance);
  return result;
}
```

В данный момент я просто делаю копию объекта спектакля, но вскоре в эту новую запись будут добавлены данные. Я делаю копию, потому что не хочу изменять данные, передаваемые в функцию. Я предпочитаю по возможности рассматривать данные как неизменяемые (константные), так как изменяемое состояние быстро приводит к проблемам.

Идиома `result=Object.assign({},aPerformance)` выглядит для тех, кто не работает с JavaScript, очень странно. Она выполняет поверхностное копирование. Я бы предпочел иметь для этого отдельную функцию, но это один из тех случаев, когда идиома настолько вросла в JavaScript, что написание отдельной функции выглядело бы неуместным для программистов на JavaScript.

Теперь, когда у меня есть место для спектакля, мне нужно его добавить. Для этого я должен применить рефакторинг *Перенос функции* (с. 244) к функциям `playFor` и `statement` (компиляция-тестирование-фиксация).

```
function statement...
function enrichPerformance(aPerformance) {
  const result = Object.assign({}, aPerformance);
  result.play = playFor(result);
  return result;
}
```

```
function playFor(aPerformance) {
  return plays[aPerformance.playID];
}
```

Затем я заменяю все обращения к `playFor` в `renderPlainText` на использование данных (компиляция-тестирование-фиксация).

```
function renderPlainText...
  let result = `Statement for ${data.customer}\n`;
  for (let perf of data.performances) {
    result += `  ${perf.play.name}: `;
    result += `${usd(amountFor(perf))} (${perf.audience} seats)\n`;
  }
  result += `Amount owed is ${usd(totalAmount())}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;

function volumeCreditsFor(aPerformance) {
  let result = 0;
  result += Math.max(aPerformance.audience - 30, 0);
  if ("comedy" === aPerformance.play.type)
    result += Math.floor(aPerformance.audience / 5);

  return result;
}

function amountFor(aPerformance) {
  let result = 0;
  switch (aPerformance.play.type) {
    case "tragedy":
      result = 40000;
      if (aPerformance.audience > 30) {
        result += 1000 * (aPerformance.audience - 30);
      }
      break;
    case "comedy":
      result = 30000;
      if (aPerformance.audience > 20) {
        result += 10000 + 500 * (aPerformance.audience - 20);
      }
      result += 300 * aPerformance.audience;
      break;
    default:
      throw new Error(`unknown type: ${aPerformance.play.type}`);
  }
  return result;
}
```

Аналогичным образом я выполняю перенос `amountFor` (компиляция-тестирование-фиксация).

```
function statement...
function enrichPerformance(aPerformance) {
  const result = Object.assign({}, aPerformance);
  result.play = playFor(result);
  result.amount = amountFor(result);
  return result;
}

function amountFor(aPerformance) {...}

function renderPlainText...
  let result = `Statement for ${data.customer}\n`;
  for (let perf of data.performances) {
    result += `  ${perf.play.name}: `;
    result += `${usd(perf.amount)} (${perf.audience} seats)\n`;
  }
  result += `Amount owed is ${usd(totalAmount())}\n`;
  result += `You earned ${totalVolumeCredits()} credits\n`;
  return result;

function totalAmount() {
  let result = 0;
  for (let perf of data.performances) {
    result += perf.amount;
  }
  return result;
}
```

Затем перемещаю расчет бонусов (компиляция-тестирование-фиксация).

```
function statement...
function enrichPerformance(aPerformance) {
  const result = Object.assign({}, aPerformance);
  result.play = playFor(result);
  result.amount = amountFor(result);
  result.volumeCredits = volumeCreditsFor(result);
  return result;
}

function volumeCreditsFor(aPerformance) {...}

function renderPlainText...
function totalVolumeCredits() {
  let result = 0;
  for (let perf of data.performances) {
    result += perf.volumeCredits;
  }
  return result;
}
```

И, наконец, перемещаю два вычисления итоговых результатов.

```
function statement...
  const statementData = {};
  statementData.customer = invoice.customer;
  statementData.performances =
    invoice.performances.map(enrichPerformance);
  statementData.totalAmount = totalAmount(statementData);
  statementData.totalVolumeCredits =
    totalVolumeCredits(statementData);
  return renderPlainText(statementData, plays);

function totalAmount(data) {...}
function totalVolumeCredits(data) {...}

function renderPlainText...
  let result = `Statement for ${data.customer}\n`;
  for (let perf of data.performances) {
    result += ` ${perf.play.name}: `;
    result += `${usd(perf.amount)} (${perf.audience} seats)\n`;
  }
  result += `Amount owed is ${usd(data.totalAmount)}\n`;
  result += `You earned ${data.totalVolumeCredits} credits\n`;
  return result;
```

Хотя можно было изменить тела этих функций так, чтобы использовать переменную `StatementData` (она находится в области видимости), я предпочитаю передачу явного параметра.

Закончив с компиляцией-тестированием-фиксацией после перемещения, я не могу устоять перед парой быстрых выстрелов рефакторинга *Замена цикла конвейером* (с. 278).

```
function renderPlainText...
function totalAmount(data) {
  return data.performances
    .reduce((total, p) => total + p.amount, 0);
}

function totalVolumeCredits(data) {
  return data.performances
    .reduce((total, p) => total + p.volumeCredits, 0);
}
```

Извлекаю весь код первой фазы в отдельную функцию (компиляция-тестирование-фиксация).

```
Верхний уровень...
function statement (invoice, plays) {
  return renderPlainText(createStatementData(invoice, plays));
}
```

```
function createStatementData(invoice, plays) {
  const statementData = {};
  statementData.customer = invoice.customer;
  statementData.performances =
    invoice.performances.map(enrichPerformance);
  statementData.totalAmount = totalAmount(statementData);
  statementData.totalVolumeCredits =
    totalVolumeCredits(statementData);
  return statementData;
```

Поскольку теперь функция явным образом отделена, я перемещаю ее в собственный файл (и изменяю имя возвращаемого результата в соответствии с моим обычным соглашением).

```
statement.js...
import createStatementData from './createStatementData.js';

createStatementData.js...
export default function createStatementData(invoice, plays) {
  const result = {};
  result.customer = invoice.customer;
  result.performances = invoice.performances.map(enrichPerformance);
  result.totalAmount = totalAmount(result);
  result.totalVolumeCredits = totalVolumeCredits(result);
  return result;

function enrichPerformance(aPerformance) {...}
function playFor(aPerformance) {...}
function amountFor(aPerformance) {...}
function volumeCreditsFor(aPerformance) {...}
function totalAmount(data) {...}
function totalVolumeCredits(data) {...}
```

Последние финальные компиляция-тестирование-фиксация — и теперь очень легко написать HTML-версию.

```
statement.js...
function htmlStatement (invoice, plays) {
  return renderHtml(createStatementData(invoice, plays));
}

function renderHtml (data) {
  let result = `<h1>Statement for ${data.customer}</h1>\n`;
  result += "<table>\n";
  result += "<tr><th>play</th><th>seats</th><th>cost</th></tr>";
  for (let perf of data.performances) {
    result += `  <tr><td>${perf.play.name}</td>`;
    result += `<td>${perf.audience}</td>`;
    result += `<td>${usd(perf.amount)}</td></tr>\n`;
  }
  result += "</table>\n";
```

```
  result += `<p>Amount owed is `;
  result += `<em>${usd(data.totalAmount)}</em></p>\n`;
  result += `<p>You earned <em>${data.totalVolumeCredits}`;
  result += `</em> credits</p>\n`;
  return result;
}

function usd(aNumber) {...}
```

(Я переместил функцию `usd` на верхний уровень, чтобы функция `renderHtml` могла ее использовать.)

Состояние: разделение на два файла (и этапы)

Сейчас хороший момент, чтобы подвести итоги и подумать о том, в каком состоянии находится код. У меня есть два файла кода.

```
statement.js...
import createStatementData from './createStatementData.js';
function statement(invoice, plays) {
  return renderPlainText(createStatementData(invoice, plays));
}

function renderPlainText(data, plays) {
  let result = `Statement for ${data.customer}\n`;
  for (let perf of data.performances) {
    result += ` ${perf.play.name}: `;
    result += `${usd(perf.amount)} (${perf.audience} seats)\n`;
  }
  result += `Amount owed is ${usd(data.totalAmount)}\n`;
  result += `You earned ${data.totalVolumeCredits} credits\n`;
  return result;
}

function htmlStatement(invoice, plays) {
  return renderHtml(createStatementData(invoice, plays));
}

function renderHtml(data) {
  let result = `<h1>Statement for ${data.customer}</h1>\n`;
  result += "<table>\n";
  result += "<tr><th>play</th><th>seats</th><th>cost</th></tr>";
  for (let perf of data.performances) {
    result += ` <tr><td>${perf.play.name}</td>`;
    result += `<td>${perf.audience}</td>`;
    result += `<td>${usd(perf.amount)}</td></tr>\n`;
  }
  result += "</table>\n";
```

```
  result += `<p>Amount owed is <em>`;
  result += `${usd(data.totalAmount)}</em></p>\n`;
  result += `<p>You earned <em>${data.totalVolumeCredits}`;
  result += `</em> credits</p>\n`;
  return result;
}

function usd(aNumber) {
return new Intl.NumberFormat("en-US",
           { style: "currency", currency: "USD",
             minimumFractionDigits: 2 }).format(aNumber/100);
}

createStatementData.js...
export default function createStatementData(invoice, plays) {
  const result = {};
  result.customer = invoice.customer;
  result.performances = invoice.performances.map(enrichPerformance);
  result.totalAmount = totalAmount(result);
  result.totalVolumeCredits = totalVolumeCredits(result);
  return result;

function enrichPerformance(aPerformance) {
  const result = Object.assign({}, aPerformance);
  result.play = playFor(result);
  result.amount = amountFor(result);
  result.volumeCredits = volumeCreditsFor(result);
  return result;
}

function playFor(aPerformance) {
  return plays[aPerformance.playID]
}

function amountFor(aPerformance) {
  let result = 0;
  switch (aPerformance.play.type) {
  case "tragedy":
    result = 40000;
    if (aPerformance.audience > 30) {
      result += 1000 * (aPerformance.audience - 30);
    }
    break;
  case "comedy":
    result = 30000;
    if (aPerformance.audience > 20) {
      result += 10000 + 500 * (aPerformance.audience - 20);
    }
    result += 300 * aPerformance.audience;
    break;
```

```
  default:
    throw new Error(`unknown type: ${aPerformance.play.type}`);
  }
  return result;
}

function volumeCreditsFor(aPerformance) {
  let result = 0;
  result += Math.max(aPerformance.audience - 30, 0);
  if ("comedy" === aPerformance.play.type)
    result += Math.floor(aPerformance.audience / 5);
  return result;
}

function totalAmount(data) {
  return data.performances
    .reduce((total, p) => total + p.amount, 0);
}

function totalVolumeCredits(data) {
  return data.performances
    .reduce((total, p) => total + p.volumeCredits, 0);
}
```

Теперь у меня больше кода, чем когда я начинал работу: более 70 строк (не считая `htmlStatement`), в отличие от около 40, — в основном из-за дополнительных "оберток", связанных с размещением кода в функциях. При прочих равных условиях большее количество кода — это плохо, но редко бывает так, чтобы все остальное оказывалось равным. Дополнительный код разбивает логику на идентифицируемые части, отделяя вычисления от создания вывода. Такая модульность облегчает понимание частей кода и того, как они сочетаются друг с другом. Краткость — это душа остроумия, а ясность — душа развивающегося программного обеспечения. Добавление модульности позволяет мне поддерживать HTML-версию кода без какого-либо дублирования вычислений.

При программировании придерживайтесь главного правила: всегда оставляйте код в более здоровом состоянии, чем он был до ваших действий.

Есть еще кое-что, что я мог бы сделать, чтобы упростить логику вывода, но пока что я не буду это делать. Всегда приходится балансировать между применением всех возможных рефакторингов и добавлением новых функций. Пока что большинство программистов уделяют рефакторингу меньше внимания, но баланс все же существует. Мое главное правило — всегда оставлять код в более здоровом состоянии, чем он был до того, как я начал рефакторинг. Код никогда не будет идеальным, но должен постоянно становиться лучше.

Реорганизация вычислений в соответствии с типом постановки

Перейдем к следующему изменению функции: поддержка большего количества категорий пьес, каждая со своими расчетами начислений и бонусов. Чтобы внести соответствующие изменения, я должен перейти к функциям расчета и отредактировать условия. Функция `amountFor` подчеркивает центральную роль, которую тип пьесы играет в выборе вычислений, но условная логика, подобная имеющейся, имеет тенденцию "портиться" по мере внесения дальнейших изменений, если она не подкреплена структурными элементами языка программирования.

Существуют различные способы добавления структурности, чтобы сделать код более явным, но в данном случае естественным подходом является полиморфизм типов — характерная особенность классической объектной ориентированности. Классическая объектная ориентированность давно является доступной (пусть и спорной) функциональной возможностью в мире JavaScript; версия ECMAScript 2015 обеспечивает ее надежный синтаксис и структуру. Поэтому имеет смысл использовать ее в такой ситуации, как рассматриваемая нами.

Мой глобальный план заключается в том, чтобы установить иерархию наследования с подклассами для комедии и трагедии, которые содержат логику расчета для этих типов пьес. Вызывающие функции будут вызывать полиморфную функцию расчета, которую язык будет диспетчеризовать в различные вычисления для комедий и трагедий. Я создам аналогичную структуру и для расчета бонусов. Для этого я использую пару рефакторингов. Основной рефакторинг *Замена условной инструкции полиморфизмом* (с. 317) заменяет часть условного кода полиморфизмом. Но прежде чем я смогу выполнить данный рефакторинг, мне нужно создать определенную структуру наследования. Необходимо создать класс для размещения функций вычисления сумм и бонусов.

Я начну с пересмотра кода вычислений. (Одним из приятных следствий предыдущего рефакторинга является то, что теперь я могу игнорировать код форматирования во время создания все той же структуры выходных данных. Я могу обеспечить дополнительную поддержку с помощью тестов, которые проверяют промежуточную структуру данных.)

```
createStatementData.js...
export default function createStatementData(invoice, plays) {
  const result = {};
  result.customer = invoice.customer;
  result.performances = invoice.performances.map(enrichPerformance);
  result.totalAmount = totalAmount(result);
  result.totalVolumeCredits = totalVolumeCredits(result);
  return result;
```

```
function enrichPerformance(aPerformance) {
  const result = Object.assign({}, aPerformance);
  result.play = playFor(result);
  result.amount = amountFor(result);
  result.volumeCredits = volumeCreditsFor(result);
  return result;
}

function playFor(aPerformance) {
  return plays[aPerformance.playID]
}

function amountFor(aPerformance) {
  let result = 0;
  switch (aPerformance.play.type) {
  case "tragedy":
    result = 40000;
    if (aPerformance.audience > 30) {
      result += 1000 * (aPerformance.audience - 30);
    }
    break;
  case "comedy":
    result = 30000;
    if (aPerformance.audience > 20) {
      result += 10000 + 500 * (aPerformance.audience - 20);
    }
    result += 300 * aPerformance.audience;
    break;
  default:
    throw new Error(`unknown type: ${aPerformance.play.type}`);
  }
  return result;
}

function volumeCreditsFor(aPerformance) {
  let result = 0;
  result += Math.max(aPerformance.audience - 30, 0);
  if ("comedy" === aPerformance.play.type)
    result += Math.floor(aPerformance.audience / 5);
  return result;
}

function totalAmount(data) {
  return data.performances
    .reduce((total, p) => total + p.amount, 0);
}

function totalVolumeCredits(data) {
  return data.performances
    .reduce((total, p) => total + p.volumeCredits, 0);
}
```

Создание калькулятора представлений

Функция enrichPerformance является ключевой, поскольку именно она заполняет промежуточную структуру данных информацией для каждого представления. В настоящее время она вызывает условные функции для вычисления суммы и бонусов. Мне нужно вызывать эти функции в принимающем классе. Поскольку этот класс содержит функции расчета данных для представлений, я назову его *калькулятором представлений* (performance calculator).

```
function createStatementData...
function enrichPerformance(aPerformance) {
  const calculator = new PerformanceCalculator(aPerformance);
  const result = Object.assign({}, aPerformance);
  result.play = playFor(result);
  result.amount = amountFor(result);
  result.volumeCredits = volumeCreditsFor(result);
  return result;
}

Верхний уровень...
class PerformanceCalculator {
  constructor(aPerformance) {
    this.performance = aPerformance;
  }
}
```

Пока что этот новый объект ничего не делает. Я хочу перенести в него поведение и начну с самого простого — записи пьесы. Строго говоря, мне не нужно этого делать, поскольку информация не будет полиморфно изменяться, но зато таким образом я соберу все преобразования данных в одном месте, и такая согласованность сделает код более понятным.

Чтобы выполнить эту работу, я использую рефакторинг *Изменение объявления функции* (с. 170) для передачи информации о представлении пьесы в калькулятор.

```
function createStatementData...
function enrichPerformance(aPerformance) {
  const calculator =
    new PerformanceCalculator(aPerformance,
                              playFor(aPerformance));
  const result = Object.assign({}, aPerformance);
  result.play = calculator.play;
  result.amount = amountFor(result);
  result.volumeCredits = volumeCreditsFor(result);
  return result;
}

class PerformanceCalculator...
class PerformanceCalculator {
```

```
  constructor(aPerformance, aPlay) {
    this.performance = aPerformance;
    this.play = aPlay;
  }
}
```

(Я больше не пишу постоянно "компиляция-тестирование-фиксация", так как подозреваю, что вы уже устали от чтения этих слов. Но я все равно выполняю компиляцию-тестирование-фиксацию при каждой возможности. Иногда я устаю это делать — и тем самым даю шанс ошибке. Это быстро приводит меня в чувство и возвращает к правильному ритму работы.)

Перемещение функций в калькулятор

Следующая часть логики, которую я перемещаю, является более существенной для вычисления суммы представления. Стоящая передо мной задача представляет собой более глубокое изменение в контексте функции, чем перемещение во вложенную функцию, поэтому я прибегну к рефакторингу *Перенос функции* (с. 244). Первая часть этого рефакторинга состоит в том, чтобы скопировать логику в ее новый контекст — класс калькулятора. Затем я подстраиваю код под его "новый дом", заменяя `aPerformance` на `this.performance` и `playFor(aPerformance)` на `this.play`.

```
class PerformanceCalculator...
get amount() {
  let result = 0;
  switch (this.play.type) {
    case "tragedy":
      result = 40000;
      if (this.performance.audience > 30) {
        result += 1000 * (this.performance.audience - 30);
      }
      break;
    case "comedy":
      result = 30000;
      if (this.performance.audience > 20) {
        result += 10000 + 500 * (this.performance.audience - 20);
      }
      result += 300 * this.performance.audience;
      break;
    default:
      throw new Error(`unknown type: ${this.play.type}`);
  }
  return result;
}
```

Сейчас я могу выполнить компиляцию, чтобы проверить наличие в коде ошибок времени компиляции. "Компиляция" в моей среде разработки происходит во время выполнения кода, поэтому на самом деле я запускаю Babel[2] — этого достаточно, чтобы перехватить любые синтаксические ошибки в новой функции, но не более того. Тем не менее это может оказаться весьма полезным шагом.

Как только новая функция удобно располагается в своем "новом доме", я беру исходную функцию и превращаю ее в делегирующую так, чтобы она вызывала новую функцию.

```
function createStatementData...
function amountFor(aPerformance) {
  return new PerformanceCalculator(aPerformance,
                                   playFor(aPerformance)).amount;
}
```

Теперь выполним компиляцию-тестирование-фиксацию, чтобы убедиться, что на новом месте код работает корректно. После этого проведем рефакторинг *Встраивание функции* (с. 161) для непосредственного вызова новой функции (компиляция-тестирование-фиксация).

```
function createStatementData...
function enrichPerformance(aPerformance) {
  const calculator =
    new PerformanceCalculator(aPerformance,
                              playFor(aPerformance));
  const result = Object.assign({}, aPerformance);
  result.play = calculator.play;
  result.amount = calculator.amount;
  result.volumeCredits = volumeCreditsFor(result);
  return result;
}
```

Повторим тот же процесс для переноса расчета бонусов.

```
function createStatementData...
function enrichPerformance(aPerformance) {
  const calculator =
    new PerformanceCalculator(aPerformance,
                              playFor(aPerformance));
  const result = Object.assign({}, aPerformance);
  result.play = calculator.play;
  result.amount = calculator.amount;
  result.volumeCredits = calculator.volumeCredits;
  return result;
}
```

[2] https://babeljs.io

```
class PerformanceCalculator...
get volumeCredits() {
  let result = 0;
  result += Math.max(this.performance.audience - 30, 0);
  if ("comedy" === this.play.type)
    result += Math.floor(this.performance.audience / 5);
  return result;
}
```

Превращение калькулятора представлений в полиморфный

Теперь, когда логика размещена в классе, пришло время применить полиморфизм. Первым шагом является использование рефакторинга *Замена кода типа подклассами* (с. 405), чтобы ввести подклассы вместо кода типа. Для этого нужно создать подклассы калькулятора представлений и использовать соответствующий подкласс в `createPerformanceData`. Чтобы получить правильный подкласс, нужно заменить вызов конструктора функцией, поскольку конструкторы JavaScript не могут возвращать подклассы. Поэтому я использую рефакторинг *Замена конструктора фабричной функцией* (с. 379).

```
function createStatementData...
function enrichPerformance(aPerformance) {
  const calculator =
    createPerformanceCalculator(aPerformance,
                                playFor(aPerformance));
  const result = Object.assign({}, aPerformance);
  result.play = calculator.play;
  result.amount = calculator.amount;
  result.volumeCredits = calculator.volumeCredits;
  return result;
}

Верхний уровень...
function createPerformanceCalculator(aPerformance, aPlay) {
  return new PerformanceCalculator(aPerformance, aPlay);
}
```

С помощью этой функции я могу создавать подклассы калькулятора производительности и использовать функцию создания объекта класса для выбора, какой из подклассов вернуть.

```
Верхний уровень...
function createPerformanceCalculator(aPerformance, aPlay) {
  switch(aPlay.type) {
  case "tragedy": return new TragedyCalculator(aPerformance,aPlay);
  case "comedy" : return new ComedyCalculator(aPerformance,aPlay);
  default:
    throw new Error(`unknown type: ${aPlay.type}`);
  }
}
```

```
class TragedyCalculator extends PerformanceCalculator {
}

class ComedyCalculator extends PerformanceCalculator {
}
```

Структура для применения полиморфизма настроена, и я могу прибегнуть к рефакторингу *Замена условной инструкции полиморфизмом* (с. 317).

Начну с вычисления суммы для трагедий.

```
class TragedyCalculator...
get amount() {
  let result = 40000;
  if (this.performance.audience > 30) {
    result += 1000 * (this.performance.audience - 30);
  }
  return result;
}
```

Просто наличия этого метода в подклассе достаточно, чтобы перекрыть условие суперкласса. Но если вы такой же параноик, как и я, вы можете сделать следующее.

```
class PerformanceCalculator...
get amount() {
  let result = 0;
  switch (this.play.type) {
    case "tragedy":
      throw 'bad thing';
    case "comedy":
      result = 30000;
      if (this.performance.audience > 20) {
        result += 10000 + 500 * (this.performance.audience - 20);
      }
      result += 300 * this.performance.audience;
      break;
    default:
      throw new Error(`unknown type: ${this.play.type}`);
  }
  return result;
}
```

Я мог бы удалить `case` для трагедии и позволить генерировать исключение ветке `default`. Но мне больше нравится явная генерация — хотя она и будет там только пару минут (именно поэтому я сгенерировал исключение со строкой, а не с объектом ошибки).

После компиляции-тестирования-фиксации я переделываю `case` для комедии.

```
class ComedyCalculator...
get amount() {
  let result = 30000;
```

```
  if (this.performance.audience > 20) {
    result += 10000 + 500 * (this.performance.audience - 20);
  }
  result += 300 * this.performance.audience;
  return result;
}
```

Теперь можно удалить метод amount суперкласса, так как он не должен вызываться никогда. Но я буду добрее и оставлю вместо него надгробную плиту.

```
class PerformanceCalculator...
get amount() {
  throw new Error('subclass responsibility');
}
```

Следующей будет выполнена замена условных выражений в расчете бонусов. Рассматривая обсуждение будущих категорий пьес, я замечаю, что большинство пьес ожидают, что аудитория превысит 30 зрителей, и только немногие категории представляют другие значения. Поэтому имеет смысл оставить более распространенный случай в суперклассе как случай по умолчанию и позволить при необходимости его переопределять.

```
class PerformanceCalculator...
get volumeCredits() {
    return Math.max(this.performance.audience - 30, 0);
}

class ComedyCalculator...
get volumeCredits() {
    return super.volumeCredits +
           Math.floor(this.performance.audience / 5);
}
```

Состояние: создание данных с помощью калькулятора представлений

Пришло время задуматься о том, что же сделал с кодом полиморфный калькулятор.

```
createStatementData.js...
export default function createStatementData(invoice, plays) {
  const result = {};
  result.customer = invoice.customer;
  result.performances = invoice.performances.map(enrichPerformance);
  result.totalAmount = totalAmount(result);
  result.totalVolumeCredits = totalVolumeCredits(result);
  return result;
```

```
  function enrichPerformance(aPerformance) {
    const calculator =
          createPerformanceCalculator(aPerformance,
                                      playFor(aPerformance));
    const result = Object.assign({}, aPerformance);
    result.play = calculator.play;
    result.amount = calculator.amount;
    result.volumeCredits = calculator.volumeCredits;
    return result;
  }

  function playFor(aPerformance) {
    return plays[aPerformance.playID]
  }

  function totalAmount(data) {
    return data.performances
      .reduce((total, p) => total + p.amount, 0);
  }

  function totalVolumeCredits(data) {
    return data.performances
      .reduce((total, p) => total + p.volumeCredits, 0);
  }
}

function createPerformanceCalculator(aPerformance, aPlay) {
  switch(aPlay.type) {
  case "tragedy": return new TragedyCalculator(aPerformance, aPlay);
  case "comedy" : return new ComedyCalculator(aPerformance, aPlay);
  default:
    throw new Error(`unknown type: ${aPlay.type}`);
  }
}

class PerformanceCalculator {
  constructor(aPerformance, aPlay) {
    this.performance = aPerformance;
    this.play = aPlay;
  }

  get amount() {
    throw new Error('subclass responsibility');
  }

  get volumeCredits() {
    return Math.max(this.performance.audience - 30, 0);
  }
}
```

```
class TragedyCalculator extends PerformanceCalculator {
  get amount() {
    let result = 40000;
    if (this.performance.audience > 30) {
      result += 1000 * (this.performance.audience - 30);
    }
  return result;
  }
}

class ComedyCalculator extends PerformanceCalculator {
  get amount() {
    let result = 30000;
    if (this.performance.audience > 20) {
      result += 10000 + 500 * (this.performance.audience - 20);
    }
    result += 300 * this.performance.audience;
    return result;
  }

  get volumeCredits() {
    return super.volumeCredits +
           Math.floor(this.performance.audience / 5);
  }
}
```

Размер кода опять увеличился, так как я ввел структуру. Преимущество в данном случае заключается в том, что расчеты для каждого вида пьес сгруппированы вместе. Если большинство изменений будут вноситься в этот код, то будет полезным четко разделить его так, как показано. Добавление пьес нового вида требует написания нового подкласса и добавления его в функцию создания.

Пример дает некоторое представление о том, когда полезно применение подклассов. Здесь я переместил условный поиск из двух функций (`amountFor` и `volumeCreditsFor`) в одну функцию конструктора `createPerformanceCalculator`. Чем больше функций зависит от одного и того же типа полиморфизма, тем более полезным становится этот подход.

Альтернативным решением был бы возврат функцией `createPerformanceData` самого калькулятора вместо заполнения калькулятором промежуточной структуры данных. Одной из приятных особенностей системы классов JavaScript является то, что в ней применение методов чтения данных ("геттеров") имеет вид обычного обращения к данным. Выбор, возвращать ли экземпляр или рассчитывать отдельные выходные данные, зависит от того, кто использует структуру данных. В данном случае я предпочел показать, как применять промежуточную структуру данных, чтобы скрыть решение использовать полиморфный калькулятор.

Заключительные замечания

Надеюсь, этот простой пример дал вам возможность почувствовать, что же такое рефакторинг. Здесь я использовал несколько видов рефакторинга, в том числе рефакторинги *Извлечение функции* (с. 152), *Встраивание переменной* (с. 169), *Перенос функции* (с. 244) и *Замена условной инструкции полиморфизмом* (с. 317).

Данный рефакторинг состоял из трех основных этапов: разложение исходной функции на набор вложенных функций, использование *Разделение этапа* (с. 201) для разделения кода вычисления и вывода и введение полиморфного калькулятора для логики расчета. Каждый из этих этапов повысил структурированность кода, позволяющую лучше понимать, что делает код.

Как это часто бывает с рефакторингом, первые его этапы были в основном вызваны попыткой понять, что происходит. Типичная последовательность действий при этом — прочесть код, получить некоторое представление о нем и использовать рефакторинг, чтобы перенести это представление из головы обратно в код. Более понятный код приводит к более глубокому пониманию и полезной петле положительной обратной связи. Конечно, есть еще некоторые улучшения, которые я мог бы сделать, но чувствую, что сделал достаточно для выполнения главного требования рефакторинга — чтобы после него код стал значительно лучше, чем был до того.

Настоящий тест на то, насколько хорош код, состоит в том, насколько легко вносить с него изменения.

Я говорю об улучшении кода, но программисты любят спорить о том, как же выглядит хороший код. Я знаю, что некоторые люди возражают против моего предпочтения маленьких функций с хорошо подобранными именами. Если считать качество кода вопросом эстетики, где нет ничего хорошего или плохого, а есть только представления о нем, то мы не сможем руководствоваться ничем, кроме личного вкуса. Однако я считаю, что мы должны выйти за рамки вкусовых предпочтений и принять точку зрения, что истинный тест на качество кода заключается в том, насколько легко его изменять. Код должен быть очевидным: если кто-то должен внести в него изменения, он должен быть в состоянии легко найти код, который нужно изменить, и внести изменения быстро и без каких-либо ошибок. Хороший код максимизирует производительность программиста, позволяя создавать больше возможностей для пользователей быстрее и дешевле. Чтобы сохранить работоспособность кода, проводите рефакторинги, приближающие код к указанному идеалу.

Но самое главное, чему можно научиться на приведенном в этой главе примере, — это ритм рефакторинга. Всякий раз, когда я показываю людям, как выполняю рефакторинг, они удивляются тому, насколько малы мои шаги. При этом каждый шаг оставляет код в рабочем состоянии — компилируемом и проходящем все тесты. Я был точно так же удивлен, когда в гостиничном номере в Детройте два десятилетия назад Кент Бек показал мне, как это делается. Ключом к эффективному рефакторингу является признание того, что вы работаете быстрее, когда делаете маленькие шаги, после которых код остается работоспособным. Вы можете объединить эти маленькие шаги в существенные изменения. В первую очередь помните именно об этом.

Глава 2

Принципы рефакторинга

Приведенный в предыдущей главе пример должен дать хорошее представление о том, что такое рефакторинг. Теперь можно сделать шаг назад и рассмотреть ключевые принципы рефакторинга и некоторые вопросы, о которых следует подумать при его выполнении.

Определение рефакторинга

Как и многие термины в разработке программного обеспечения, термин *рефакторинг* часто используется практикующими программистами очень вольно. Я считаю, что данный термин следует использовать в более точной форме. (Приведенные определения такие же, как те, что я дал в первом издании этой книги.) Термин *рефакторинг* можно использовать как для процесса, так и для результата. Определение результата таково.

> **Рефакторинг**: *изменения внутренней структуры программного обеспечения, позволяющие облегчить понимание его работы и упростить модификацию без изменения наблюдаемого поведения.*

Это определение рефакторинга соответствует таким рассмотренным ранее примерам, как рефакторинги *Извлечение функции* (с. 152) или *Замена условной инструкции полиморфизмом* (с. 317).

Определение термина *рефакторинг* для процесса.

> **Рефакторинг**: *внесение изменений в структуру программного обеспечения с помощью ряда преобразований, не затрагивающих поведение модифицируемого программного обеспечения.*

Таким образом, можно в течение часов заниматься рефакторингом как процессом и при этом выполнить пару дюжин рефакторингов-результатов.

За прошедшие годы многие программисты стали использовать термин *рефакторинг* для обозначения любого вида улучшения кода, но приведенные выше определения указывают на особый подход к улучшению кода. Рефакторинг — это применение небольших шагов, сохраняющих поведение, и внесение больших изменений путем связывания воедино последовательности упомянутых малых шагов, сохраняющих поведение. Каждый отдельный рефакторинг сам по себе либо довольно маленький, либо представляет собой комбинацию маленьких шагов. В результате, когда я выполняю рефакторинг, мой код не оказывается надолго в неисправном состоянии, что позволяет мне останавливаться в любой момент, даже если я еще не закончил всю намеченную работу полностью.

Если кто-то говорит, что его код был неработоспособен в течение нескольких дней во время выполнения рефакторинга — можете быть уверены, что никакого рефакторинга не было.

Я использую в качестве общего термина для обозначения любого вида реорганизации или улучшения кода термин *реструктуризация* и рассматриваю рефакторинг как особый подвид реструктуризации. Рефакторинг может показаться неэффективным людям, которые впервые сталкиваются с ним и смотрят, как я делаю много маленьких шагов там, где можно было бы сделать один большой шаг. Но крошечные шаги позволяют мне идти быстрее, потому что они хорошо сочетаются, и, что особенно важно, потому что я не трачу время на отладку.

В моих определениях я использую фразу "наблюдаемое поведение". Это намеренно вольный термин, указывающий, что код должен делать в целом то же, что и до того, как я начал работу. Это не означает, что он будет работать абсолютно так же, как и ранее. Например, рефакторинг *Извлечение функции* (с. 152) изменяет стек вызовов, поэтому характеристики производительности могут измениться, но не должно измениться ничего, о чем должен волноваться пользователь. В частности, интерфейсы модулей часто изменяются из-за таких рефакторингов, как *Изменение объявления функции* (с. 170) и *Перенос функции* (с. 244). Любые ошибки, которые я замечаю во время рефакторинга, должны присутствовать и после рефакторинга (хотя я могу исправить скрытые ошибки, которые еще никто не наблюдал).

Рефакторинг очень похож на оптимизацию производительности, поскольку они оба включают в себя выполнение манипуляций с кодом, не изменяющие общую функциональность программы. Разница заключается в цели: рефакторинг всегда делается для того, чтобы сделать код "более легким для понимания и более простым для изменений". Но это может как ускорить, так и замедлить работу кода. При оптимизации производительности я забочусь только об ускорении программы и готов закончить работу со сложным кодом, который действительно обеспечивает столь нужную мне улучшенную производительность.

Две шляпы

Кент Бек познакомил меня с метафорой двух шляп. Используя рефакторинг при разработке программного обеспечения, вы делите рабочее время между двумя разными видами деятельности: добавлением функциональности и рефакторингом. При добавлении новой функциональности вы не должны менять структуру существующего кода; вы просто добавляете новые возможности. Достигаемый результат можно оценить, добавляя тесты и добиваясь их нормальной работы. При проведении рефакторинга вы стараетесь не добавлять функциональность, а только улучшать структуру исходных текстов. При этом новые тесты не добавляются (разве что в случае обнаружения пропущенного ранее тестового случая); изменяются тесты только тогда, когда это совершенно необходимо для проверки изменений в интерфейсе.

При разработке программного обеспечения бывает необходимо часто переключаться между указанными видами деятельности. Пытаясь добавить новую функциональность, можно выяснить, что если изменить саму структуру кода, то добиться этого будет куда легче. В таком случае нужно на время снять шляпу разработчика и надеть шляпу специалиста по рефакторингу. Изменив структуру кода, можно вновь надеть шляпу программиста и добавить новую функциональность. Добившись ее работоспособности, можно заметить, что ее трудно понять, и тогда следует снова сменить шляпу и заняться рефакторингом. Эти роли могут меняться буквально каждые десять минут, но в каждый момент времени вы должны ясно отдавать себе отчет, какой именно деятельностью вы сейчас заняты.

Почему нужно заниматься рефакторингом

Я не стану утверждать, что рефакторинг — лекарство от всех болезней. Это не панацея. Это просто очень ценный инструмент — некие "серебряные плоскогубцы", позволяющие надежно вцепиться в код. Рефакторинг представляет собой инструмент, который можно и нужно использовать для нескольких целей.

Рефакторинг совершенствует проектирование программного обеспечения

Без рефакторинга внутренний проект программного обеспечения — его архитектура — имеет тенденцию к ухудшению. Поскольку люди изменяют код для достижения краткосрочных целей, часто без полного понимания архитектуры, код теряет свою структурированность. Становится все сложнее увидеть проект, просто читая код. Утрата структуры кода имеет кумулятивный эффект. Чем сложнее

увидеть проект в коде, тем сложнее его сохранить и тем быстрее он распадается. Регулярный рефакторинг помогает поддерживать код в форме.

Плохо спроектированное программное обеспечение обычно требует большего кода, зачастую просто потому, что этот код выполняет в разных местах одни и те же действия. Поэтому важной частью улучшения проекта является устранение дублирования кода. Важность этого связана с будущими изменениями кода. Уменьшение кода не делает программу более быстрой, так как размер программы в памяти редко играет большую роль. Однако размер кода играет роль при внесении в него изменений. Чем больше код, тем труднее корректно внести в него изменения, а чтобы понять, как он работает, приходится разбираться в большом количестве строк. При внесении изменений в код в одном месте программа может вести себя не так, как предполагалось, поскольку не внесены изменения в некоторый другой участок, который выполняет те же действия, но в несколько ином контексте. Устраняя дублирование, мы гарантируем, что код содержит все, что нужно, причем в единственном месте, — а в этом и состоит суть хорошего проектирования.

Рефакторинг упрощает понимание программ

Во многом программирование напоминает общение с компьютером. Программист пишет код, который указывает компьютеру, что тот должен делать, и в ответ компьютер точно следует указаниям. Постепенно вы уменьшаете разрыв между тем, что должен делать компьютер, и тем, что вы ему говорите. Таким образом, суть программирования сводится к тому, чтобы абсолютно точно указать компьютеру, что от него требуется. Но программа предназначена не только для компьютера. Пройдет некоторое время, и кому-нибудь может понадобиться ваш исходный текст, чтобы внести некоторые изменения. Об этом будущем программисте часто забывают, но он, по сути, и есть главный пользователь вашего исходного текста (не программы, а именно исходного текста). Вряд ли кого-то сильно обеспокоит несколько дополнительных минут для компиляции, но будет плохо, если программист потратит неделю на внесение изменений, которые заняли бы один час, если бы он мог сразу же разобраться в коде.

Проблема в том, что, пытаясь заставить программу работать, вы совсем не думаете о тех, кто будет заниматься ею в будущем. Внесение в код изменений, облегчающих его понимание, меняет ритм работы. Рефакторинг делает код более простым для чтения и понимания. При выполнении рефакторинга вы берете работоспособный код, который не отличается красотой структуры. Затратив немного времени на рефакторинг, можно добиться лучшей самодокументируемости кода. Теперь суть программирования сводится к тому, чтобы абсолютно точно сказать, что вы имеете в виду.

Такой подход связан не только с альтруизмом. Этим будущим разработчиком часто бываю я сам. И тогда рефакторинг приобретает особую важность. Я очень ленив. В частности, моя лень проявляется в том, что я не запоминаю деталей кода, который пишу. Более того, опасаясь перегрузить свою голову, я умышленно не запоминаю многие детали. И поэтому я стараюсь записывать в коде все, что иначе мне пришлось бы держать в моей бедной голове. Это позволяет мне не беспокоиться о воздействии пары бокалов пива после работы на клетки моего головного мозга.

Рефакторинг помогает находить ошибки

Чем яснее я понимаю код, тем проще мне найти в нем ошибки. Следует сказать, что я не мастак искать ошибки. Если другие ухитряются, просмотрев большой фрагмент кода, тут же увидеть в нем дефекты, то для меня это недостижимая мечта. Но я заметил, что когда я выполняю рефакторинг, мне приходится глубоко вникать в код и разбираться, что и как он делает. Это понимание возвращается в код. После структуры программы я проясняю для себя некоторые высказанные мною предположения и в результате просто не могу пропустить ошибки.

Это напоминает мне высказывание Кента Бека, которое он часто повторяет: "Я не слишком хороший программист. Но я программист с хорошими привычками". Рефакторинг является такой хорошей привычкой, которая помогает мне писать надежный код.

Рефакторинг ускоряет написание программ

В конечном итоге все сказанное сводится к одному: рефакторинг ускоряет написание программ.

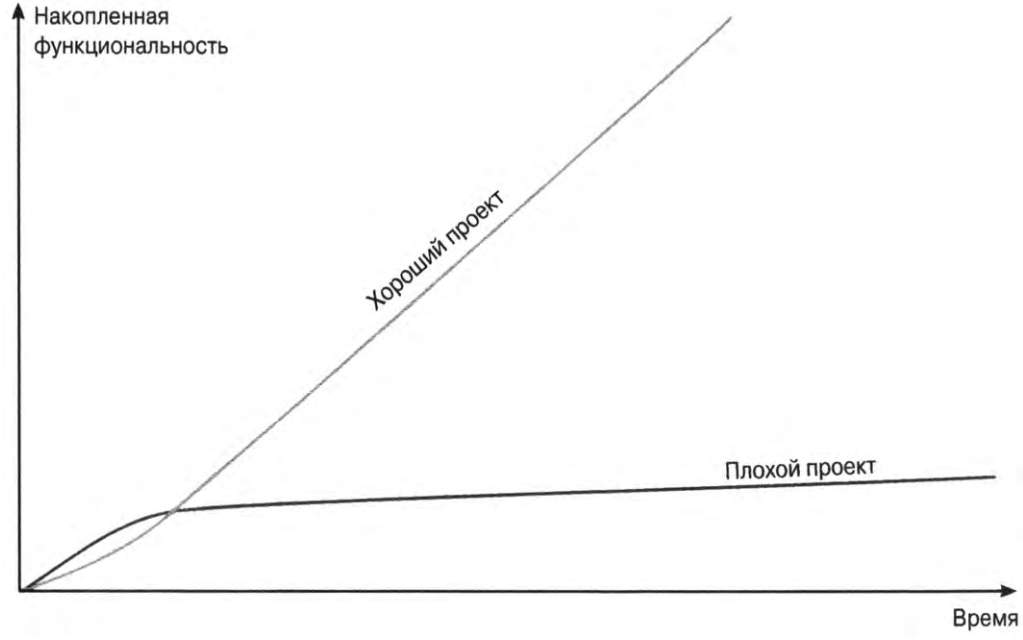
Создается впечатление внутреннего противоречия. Когда я рассказываю о рефакторинге, становится очевидно, что он повышает качество кода. Улучшение проекта, повышение удобочитаемости, уменьшение количества ошибок — все это способствует качеству кода. Но разве скорость разработки не снижается из-за всего этого?

Когда я общаюсь с разработчиками программного обеспечения, которые какое-то время работали над системой, я часто слышу, что сначала им удалось быстро продвинуться вперед, но теперь добавление новых функциональных возможностей занимает гораздо больше времени. Каждая новая функция требует все больше и больше времени, чтобы понять, как вписать ее в существующую кодовую базу, а после ее добавления часто возникают ошибки, исправление которых занимает еще больше времени. Кодовая база начинает выглядеть как серия исправлений, исправляющих предыдущие исправления, и требуются навыки археолога,

чтобы выяснить, как все это работает. Все это замедляет добавление новых функциональных возможностей до такой степени, что зачастую разработчики хотят начать все заново с чистого листа.

Визуализировать это положение вещей можно с помощью следующего псевдографика.

Но некоторые команды сообщают о другом опыте. Они утверждают, что могут добавлять новые функциональные возможности *быстрее*, потому что они могут использовать уже существующий код, опираясь на то, что уже имеется в наличии.



Разница между этими проектами заключается во внутреннем качестве программного обеспечения. Программное обеспечение с хорошим внутренним проектом позволяет легко найти, какие нужно внести изменения, чтобы добавить новую функциональную возможность, и где. Хорошая модульность позволяет понять только небольшое подмножество кода, в которое нужно вносить изменения. Если код понятен, меньше вероятность внести ошибку, а если это и произойдет, процесс отладки будет намного проще. Так кодовая база превращается в платформу для создания новых функциональных возможностей для своей предметной области.

Я называю этот эффект *гипотезой стойкости проекта* (Design Stamina Hypothesis [22]): создавая хороший внутренний проект, мы повышаем стойкость программного обеспечения, позволяющую двигаться быстрее. Я не могу доказать, что это так, поэтому называю это утверждение гипотезой. Но так подсказывает мой опыт, а также опыт сотен отличных программистов, с которыми я познакомился за свою карьеру.

Двадцать лет назад общепринятым было мнение, что для создания хорошего проекта нужно завершить проектирование до начала кодирования, потому что, как только мы написали код, мы можем столкнуться только с ухудшением и упадком. Рефакторинг меняет эту картину. Теперь мы знаем, что можем улучшить проект существующего кода, так что мы можем формировать и улучшать проект с течением времени, даже когда меняются потребности программы. Поскольку очень сложно сделать хороший проект заранее, рефакторинг становится жизненно важным.

Когда нужно выполнять рефакторинг

Рефакторинг — это та деятельность, которой я занимаюсь при программировании ежечасно.

Правило трех раз

Дон Робертс (Don Roberts) однажды сказал мне следующее. Когда вы делаете что-то в первый раз, вы просто это делаете. Во второй раз вы морщитесь от повторения тех же действий, но все же делаете их. Наконец, делая это же в третий раз, вы начинаете рефакторинг.

Начинайте рефакторинг после трех повторов.

Подготовительный рефакторинг — упрощение добавления функциональной возможности

Лучшее время для рефакторинга — непосредственно перед добавлением в код новой функциональной возможности. Делая это, я смотрю на существующий код и часто вижу, что если бы он был структурирован по-другому, моя работа была бы намного проще. Возможно, есть функция, которая делает почти все, что мне нужно, но содержит некоторые литеральные значения, противоречащие моим потребностям. Без рефакторинга я мог бы скопировать данную функцию и изменить эти значения. Но это приведет к дублированию кода — если мне нужно будет изменить его в будущем, то придется изменить оба места (и, что еще хуже, найти их). Копирование-вставка не поможет, если мне понадобится сделать аналогичный вариант для новой функции в будущем. Так что, надев шляпу рефакторинга, я использую рефакторинг *Параметризация функции* (с. 355). Как только я сделаю все, что нужно, мне останется лишь вызвать функцию с требуемыми параметрами.

> *"Выглядит так, как будто мне нужно пройти 100 миль на восток, но вместо того, чтобы просто продираться через лес, я собираюсь проехать 20 миль на север к шоссе, а затем проехать 100 миль на восток, добравшись в три раза быстрее, чем если бы я просто отправилась прямо туда. Когда люди подталкивают вас идти по самому прямому маршруту, иногда стоит не спешить, а ознакомиться с картой и найти самый быстрый маршрут. Именно это и делает подготовительный рефакторинг".*
>
> *— Джессика Керр (Jessica Kerr),*
> `https://martinfowler.com/articles/preparatory-refactoring-example.html`

То же самое происходит и при исправлении ошибки. Обычно, как только причина проблемы найдена, я вижу, что ее было бы намного легче исправить, если бы я объединил три фрагмента скопированного кода, вызывающего ошибку, в один. Или, возможно, отделение некоторой логики обновления от запросов поможет избежать путаницы, вызывающей эту ошибку. Выполняя рефакторинг для улучшения ситуации, я не только исправляю ошибку, но и сокращаю шансы возникновения других ошибок в тех же местах кода.

Осмыслительный рефакторинг — упрощение понимания кода

Прежде чем я смогу изменить какой-то код, я должен понять, что он делает. Возможно, этот код был написан мной или кем-то иным. Всякий раз, когда мне приходится напряженно думать, чтобы понять, что делает код, я спрашиваю себя,

не могу ли я реорганизовать код так, чтобы сделать это понимание более очевидным. Возможно, это была некоторая беспорядочная условная логика. А может быть, я хотел использовать некоторые существующие функции, но потратил несколько минут, чтобы выяснить, что они делают, потому что им дали плохие имена.

В некоторый момент у меня в голове появляется некоторое понимание, но моя голова — не очень хорошая записная книжка для таких деталей. Как говорит Уорд Каннингем (Ward Cunningham), путем рефакторинга я перемещаю понимание из своей головы в сам код. Затем я проверяю это понимание, запуская программное обеспечение, чтобы увидеть, работает ли оно по-прежнему. Если я перенесу свое понимание в код, оно сохранится там и далее и будет видно моим коллегам.

Это не просто помогает мне в будущем — зачастую это помогает мне прямо сейчас. Вначале я выполняю осмыслительный рефакторинг над мелкими деталями. Я переименовываю пару переменных, чтобы понять, что они собой представляют, или делю длинную функцию на более мелкие части. Затем, когда код становится более понятным, я обнаруживаю, что могу видеть в проекте то, чего не видел раньше. Если бы я не изменил код, то, вероятно, никогда бы не увидел этих вещей, просто потому что я недостаточно умен, чтобы визуализировать все это в своей голове. Ральф Джонсон (Ralph Johnson) описывает эти ранние рефакторинги как стирание грязи с окна, чтобы вы могли видеть лучше и дальше. Когда я изучаю код, рефакторинг выводит меня на более высокий уровень понимания, который иначе был бы мною пропущен. Те, кто отвергают осмыслительный рефакторинг как бесполезную игру с кодом, не понимают, что, отказавшись от него, они никогда не увидят возможности, скрытые путаницей в коде.

Убирающий рефакторинг

Вариацией осмыслительного рефакторинга является ситуация, когда я понимаю, что делает код, но понимаю также, что он делает это плохо — излишне запутанная логика, почти идентичные функции, которые могут быть заменены одной параметризованной функцией. Здесь есть определенный компромисс. Я не хочу тратить много времени, отвлекаясь от задачи, которую выполняю прямо сейчас, но не хочу и оставлять мусор валяться под ногами и мешать будущим изменениям. Если ситуацию легко изменить — я делаю это сразу же. Если исправление требует немного больше времени и усилий — я могу вернуться и исправить проблему, когда выполню свою непосредственную задачу.

Иногда исправление требует несколько часов, а у меня есть более срочные дела. Однако даже в этом случае, как правило, стоит сделать код хоть немного лучше. Всегда оставляйте код более чистым, чем когда вы его нашли. Если я сделаю его немного лучше каждый раз, когда прохожу через него, со временем он исправится.

Что хорошо в рефакторинге — так это то, что я не нарушаю работоспособность кода на каждом маленьком шаге, поэтому, если для завершения работы требуются месяцы, код не теряет работоспособность, даже когда я нахожусь на полпути.

Запланированный и спонтанный рефакторинги

Приведенные выше примеры рефакторингов — спонтанные. Я не выделяю время для рефакторинга — я выполняю его как часть добавления функции или исправления ошибки. Это часть моей естественной работы программиста. Независимо от того, добавляю я функцию или исправляю ошибку, рефакторинг помогает мне выполнить непосредственную задачу, а также позволяет облегчить будущую работу. Это важный момент, который часто упускается. Рефакторинг — это деятельность, которая неотделима от программирования. Я не уделяю времени своим планам по рефакторингу; большинство рефакторингов выполняются, пока я занимаюсь другими делами.

Вы должны проводить рефакторинг, когда сталкиваетесь с уродливым кодом, но и отличному коду тоже требуется рефакторинг.

Распространенной ошибкой является представление, что рефакторинг — это то, что люди делают для исправления прошлых ошибок или исправления некрасивого кода. Конечно, вам придется проводить рефакторинг, когда вы сталкиваетесь с уродливым кодом, но отличный код тоже нуждается в большом количестве рефакторинга. Всякий раз, когда я пишу код, я выбираю — сколько мне нужно параметров, где проводить черты между функциями. Выбор, правильный для вчерашнего набора функций, может не подходить для новых функций, добавляемых сегодня. Преимущество рефакторинга заключается в том, что чистый код легче реорганизовать, когда мне нужно изменить эти параметры, отражая новую реальность.

> *"Сделайте каждое желаемое изменение легким (предупреждение: это может быть трудно), затем проведите это легкое изменение".*
>
> *— Кент Бек,*
>
> `https://twitter.com/kentbeck/status/250733358307500032`

В течение долгого времени люди рассматривали создание программ как процесс наращивания: чтобы добавить новые функции, мы должны в основном добавлять новый код. Но хорошие разработчики знают, что часто самый быстрый способ добавить новую функцию — это изменить код так, чтобы ее было легко добавлять. Таким образом, программное обеспечение никогда не следует считать "завершенным". Поскольку постоянно необходимы новые возможности,

программное обеспечение изменяется соответствующим образом, отражающим этот факт. Эти изменения часто могут проявляться больше в существующем коде, чем в новом.

Все это не означает, что планировать рефакторинг — всегда неверно. Если команда пренебрегала рефакторингом, в результате ей зачастую требуется отдельное выделенное время, чтобы привести свою кодовую базу в состояние, пригодное для добавления новых функций, и неделя, проведенная за рефакторингом, может окупиться в течение следующих нескольких месяцев. Иногда, даже при регулярном рефакторинге, я вижу, как проблемная область разрастается до такой степени, что для решения проблем требуются согласованные усилия. Но такие запланированные эпизоды рефакторинга должны быть редкими. Большинство усилий по рефакторингу должны быть ничем не примечательными и спонтанными.

Один из советов, который я слышал, состоит в том, чтобы разделять работы по рефакторингу и добавлению новых функций в разных фиксациях кода в системах контроля версий. Большим преимуществом этого решения является то, что они могут быть рассмотрены и утверждены независимо. Однако я не уверен в этом решении — слишком часто рефакторинги тесно переплетаются с добавлением новых функций, и тратить время на их разделение не стоит. Такой подход может также удалить контекст рефакторинга, что затруднит обоснование фиксаций кода рефакторинга. Каждая команда должна экспериментировать, чтобы найти то решение, которое лучше всего работает для них. Просто помните, что отделение фиксаций кода рефакторинга не является самоочевидным принципом — оно имеет смысл только в том случае, если облегчает жизнь разработчикам.

Долгосрочный рефакторинг

Большая часть рефакторингов может быть завершена в течение нескольких минут, максимум — часов. Но существуют рефакторинги, которые могут занять несколько недель работы всей команды. Возможно, потребуется заменить существующую библиотеку новой. Или извлечь некоторый фрагмент кода в компонент, которым они смогут поделиться с другой командой. Или исправить какой-то неприятный беспорядок зависимостей, которые они позволили себе создать.

Но даже в таких случаях я не хочу, чтобы команда выполняла специальный рефакторинг. Часто полезной стратегией является согласие на постепенную работу над проблемой в течение следующих нескольких недель. Всякий раз, когда кто-нибудь подходит к какому-либо коду, находящемуся в зоне рефакторинга, он немного перемещает его в направлении улучшения. Здесь используется тот факт, что рефакторинг не нарушает работоспособность кода — каждое небольшое изменение оставляет весь код в работоспособном состоянии. Чтобы перейти от одной библиотеки к другой, начните с введения новой абстракции, которая может

выступать в качестве интерфейса к любой библиотеке. Как только вызывающий код станет использовать эту абстракцию, будет гораздо проще перейти от одной библиотеки к другой. (Эта тактика называется "Ветвление с помощью абстракции" [18].)

Рефакторинг в ходе анализа кода

В ряде организаций регулярно проводятся собрания по анализу кода. В других этим не занимаются — и совершенно напрасно. Анализ, или обзор, кода способствует распространению знаний среди всей команды разработчиков. Более опытные разработчики таким образом делятся своими знаниями с менее опытными. Такие встречи для анализа кода помогают большему количеству программистов понять большее число аспектов крупной программной системы. Они очень важны для написания понятного кода. Код, написанный мною, может казаться понятным мне, но не моей команде. Это нормально, ведь очень трудно поставить себя на место того, кто не знает, как вы работаете. Анализ позволяет также высказать полезные мысли большему количеству людей. У одного человека — особенно такого, как я, — столько разных хороших идей не появится и за неделю. Вклад всех облегчает жизнь каждого, поэтому лично я всегда стараюсь почаще посещать такие собрания.

Оказалось, рефакторинг помогает мне разбираться в коде, написанном другими программистами. До того, как я стал активно использовать рефакторинг, я мог прочесть код, в определенной мере понять его и внести свои предложения по его улучшению. Теперь же, когда у меня возникают идеи, я сразу решаю, нельзя ли их немедленно реализовать с помощью рефакторинга. Если можно, то я выполняю рефакторинг. Проделав так несколько раз, я могу более ясно увидеть, как будет выглядеть код после внесения в него предлагаемых изменений. Мне не надо напрягать воображение, чтобы представить будущий код, — я сразу вижу его. В результате я прихожу к идеям следующего уровня, которые без рефакторинга не появились бы.

Кроме того, рефакторинг помогает получать более конкретные результаты от анализа кода. При его выполнении новые предложения не только возникают, но в основном тут же и реализуются. В результате мероприятие оказывается куда более успешным, чем было бы без рефакторинга.

Как именно следует вводить рефакторинг в обзор кода, зависит от характера обзора. Обычная модель, когда рецензент просматривает код без первоначального автора, работает не слишком хорошо. Лучше иметь рядом автора кода, потому что автор может предоставить контекст кода и полностью оценить намерения рецензентов в плане их изменений. Мой наилучший опыт — это когда я работал в паре с автором кода, просматривал исходные тексты и выполнял рефакторинг

по ходу дела. Логическим следствием этого стиля является *парное программирование*: непрерывный обзор кода, встроенный в сам процесс программирования.

Что мне сказать руководству?

Мне очень часто задают вопрос "Как пояснить важность рефакторинга руководству?" Мне попадались места, где слово "рефакторинг" в устах менеджеров было ругательством — ведь с их точки зрения рефакторинг — не что иное, как исправление ошибок, которых при аккуратном программировании просто не должно быть! Ситуация усугубляется, если команды планируют недели чистого рефакторинга — особенно если на самом деле это не рефакторинг, а менее тщательная реструктуризация, которая вызывает неработоспособность кода.

Менеджеру, который действительно разбирается в технологиях и понимает гипотезу стойкости проекта, оправдать рефакторинг не трудно. Такие менеджеры сами поощряют рефакторинг на регулярной основе и ищут признаки, которые указывают, что команда занимается им недостаточно. Хотя, случается, что команды делают слишком много рефакторинга, но это происходит гораздо реже, чем когда команды делают его достаточно.

Конечно, часто руководители лишь говорят, что качество для них — самое главное, но на деле их беспокоит только график работ. В этом случае мой совет несколько спорный: *не говорите им вообще ничего*!

Саботаж? Вовсе нет. Мы, разработчики программного обеспечения, — профессионалы, и наша работа заключается в наиболее быстром создании эффективного программного обеспечения. Мой опыт показывает, что рефакторинг существенно повышает скорость создания приложений. Когда нужно добавить новую функцию, а проект при этом не позволяет выполнить эту модификацию быстро и просто, то быстрее будет изменить структуру проекта, а уж после добавлять новую функциональность. Если нужно исправить ошибку, то первое, что нужно, — это понять, как работает программа, а быстрее всего этого можно добиться с помощью рефакторинга. Руководитель, подгоняемый графиком работ, хочет от меня повышения скорости; ну, а как именно я буду этого добиваться — не его дело. Самый быстрый путь — рефакторинг, а потому я буду им заниматься.

Когда не следует прибегать к рефакторингу

Может показаться, что я всегда рекомендую рефакторинг, но бывают случаи, когда прибегать к нему не стоит.

Если я сталкиваюсь с беспорядочным кодом, но мне не нужно его изменять, — я не буду прибегать к его рефакторингу. Некоторый уродливый код, который можно рассматривать как API, может оставаться безобразным. Рефакторинг дает выгоду только тогда, когда мне нужно понять, как этот код работает.

Еще один пример — когда код настолько запутан, что его проще переписать заново, чем подвергнуть рефакторингу. Такое решение принять нелегко, оно требует участия здравого смысла и опыта, и я признаюсь, что не могу дать какие-либо надежные рекомендации по этому поводу.

Проблемы при рефакторинге

Всякий раз, когда кто-то выступает за какую-то методику, инструмент или архитектуру, я всегда ищу проблемы. У всего и всегда есть какие-то минусы. Следует их понимать, чтобы решить, когда и где применять тот или иной способ. Я действительно думаю, что рефакторинг — это ценная технология, которая должна использоваться большинством команд. Но с ней связаны свои проблемы, и важно понимать, как и в чем они проявляются, и как можно на них реагировать.

Замедление внедрения новых возможностей

Если вы читали предыдущий раздел, вы уже должны знать мой ответ. Хотя многие люди считают, что время, потраченное на рефакторинг, замедляет разработку новых функций, основная цель рефакторинга как раз в том, чтобы ускорить этот процесс. Но хотя это и так, верно также и то, что восприятие рефакторинга как замедляющего действия все еще широко распространено и, возможно, является самым большим препятствием для выполнения разработчиками рефакторинга в достаточном количестве.

Главная цель рефакторинга — позволить нам программировать быстрее, помогая создавать больше ценой меньших усилий.

Я часто сталкиваюсь с ситуациями, когда необходимо провести масштабный рефакторинг, но новая функциональная возможность, которую я хочу добавить, настолько мала, что я предпочитаю добавить ее и оставить большой рефакторинг на потом. Это часть моих профессиональных навыков программиста, и я не в состоянии легко описать (не говоря уже о количественной оценке), как я выполняю анализ и прихожу к такому выводу.

Я прекрасно понимаю, что подготовительный рефакторинг часто облегчает внесение изменений, и поэтому я обязательно выполню его, — если увижу, что он облегчает реализацию новой функциональной возможности. Я также более склонен к рефакторингу, если это проблема, с которой я уже сталкивался раньше. Иногда мне требуется увидеть какое-то конкретное безобразие несколько раз, прежде чем я решусь на рефакторинг. С другой стороны, я, скорее всего, не буду

проводить рефакторинг, если это та часть кода, к которой я прикасаюсь крайне редко. Иногда я откладываю рефакторинг, потому что не уверен, какое именно улучшение следует выполнить, хотя в других случаях я могу попробовать что-то в качестве эксперимента, просто чтобы посмотреть, не улучшит ли это ситуацию.

Тем не менее свидетельства, которые я часто слышу от своих коллег по отрасли, подтверждают, что слишком мало рефакторинга встречается гораздо чаще, чем слишком много. Другими словами, подавляющее большинство разработчиков должны пытаться прибегать к рефакторингу чаще.

Хотя менеджеров часто критикуют за непродуктивную привычку пренебрегать рефакторингом во имя скорости, я часто видел, как это делают разработчики. Иногда они полагают, что им не следует заниматься рефакторингом, несмотря на то, что их руководство на самом деле его поддерживает. Если вы технический руководитель команды, важно показать ее членам, что вы цените улучшение работоспособности кода. Члены команды с малым опытом нуждаются в наставничестве, которое позволит им ускорить процесс обучения и выполнения рефакторинга.

Но я все же думаю, что самая опасная ловушка — это когда разработчики пытаются оправдать рефакторинг с точки зрения "чистого кода", "хорошей инженерной практики" или аналогичных моральных соображений. Смысл рефакторинга не в том, чтобы показать, насколько блестящим является код. Рефакторинг — это чисто прагматичный подход. Мы проводим рефакторинг, потому что он позволяет нам работать быстрее — быстрее добавлять функциональные возможности, быстрее исправлять ошибки. Важно помнить об этом самому и напоминать другим. Движущим фактором должны быть экономические выгоды рефакторинга, и чем лучше это понимают разработчики, менеджеры и клиенты, тем более крутую кривую "хорошего проекта" мы увидим.

Владение кодом

Многие рефакторинги включают внесение изменений, которые влияют не только на внутренние компоненты модуля, но и на его взаимоотношения с другими частями системы. Если я хочу переименовать функцию и могу найти все вызовы этой функции, я просто применяю рефакторинг *Изменение объявления функции* (с. 170), меняя объявление и вызывающие функции за одно изменение. Но иногда такой простой рефакторинг невозможен. Например, если вызывающий код принадлежит другой команде разработчиков, и у меня нет прав на запись в их хранилище кода. Возможно, эта функция — объявленный API, используемый клиентами, поэтому я даже не могу сказать, используется ли он, не говоря уже о том, кем и как часто. Такие функции являются частью **опубликованного интерфейса** — т.е. интерфейса, который используется клиентами, независимыми от разработчиков интерфейса.

Границы владения кодом мешают проведению рефакторинга, потому что я не могу вносить все изменения, которые хочу, не нарушая работу моих клиентов. Это не запрещает рефакторинг как таковой — я все еще могу многое сделать — но налагает на него ограничения. При переименовании функции мне нужно не только использовать рефакторинг *Изменение объявления функции* (с. 170), но и сохранить старое объявление как переход к новому. Это усложняет интерфейс, но данную цену я должен заплатить, чтобы не испортить жизнь клиентов. Я могу пометить старый интерфейс как не рекомендованный к употреблению и со временем удалить его, но зачастую такой интерфейс приходится сохранять навсегда.

Из-за этих сложностей я выступаю против мелкозернистого владения кодом. Некоторым организациям нравится, когда любой фрагмент кода имеет одного программиста в качестве владельца, и только этот программист имеет право вносить в данный код изменения. Я видел, как команда из трех человек работала таким образом, что каждый из них публиковал интерфейсы для двух других. Это вело ко всевозможным сложностям в поддержке интерфейсов в ситуациях, когда было бы намного внести нужные изменения в базу кода. Я предпочитаю разрешить владеть кодом команде в целом, чтобы любой член команды мог изменить код, даже если изначально он был написан кем-то другим. Программисты могут нести личную ответственность за некоторые области системы, но это должно подразумевать, что они следят за изменениями в своей области ответственности, а не блокируют их по умолчанию.

Такая более разрешительная схема владения может распространяться даже между командами. Некоторые команды рекомендуют модель с открытым исходным кодом, в которой люди из других команд могут изменить ветвь своего кода и отправить фиксацию кода на утверждение. Это позволяет одной команде изменить клиентов своих функций — теперь они смогут удалить старые объявления, как только их изменения клиентов будут приняты. Часто это может быть хорошим компромиссом между схемой сильного владения кодом и хаотическими изменениями в больших системах.

Ветви

Сейчас, когда я пишу эти строки, распространенный подход в командах разработчиков заключается в том, что каждый член команды работает над ветвью кодовой базы, используя систему контроля версий, и выполняет значительную работу над этой ветвью, прежде чем поделиться ею, интегрировав ее с общей (главной, или магистральной) линией. Часто это включает построение целой функциональной возможности в ветви, без включения в основную линию, пока эта возможность не будет полностью готова к выпуску в производственной версии. Поклонники такого подхода утверждают, что он очищает основную линию от любого внутреннего кода, обеспечивает точную историю версий добавлений

функциональных возможностей и позволяет легко выполнять откат в случае возникновения проблем.

У этого подхода есть свои недостатки. Чем дольше я работаю в изолированной ветви, тем сложнее будет интегрировать мою работу с основной линией, когда я закончу. Большинство людей уменьшают остроту этой проблемы путем частого слияния или переноса кода из основной линии в свои ветви. Но это не решает проблему, когда над отдельными ветвями работают несколько человек. Я различаю слияние и интеграцию. Если я объединяю основную линию с моим кодом, это одностороннее движение: моя ветвь меняется, а основная — нет. Я использую термин *интеграция* для описания двустороннего процесса, который переносит изменения из основной линии в мою ветвь, а затем возвращает результат в основную линию, тем самым изменяя и ту, и другую. Если Рэйчел работает над своей ветвью, я не увижу ее изменения, пока она не интегрируется с основной линией; после этого я должен внести ее изменения в свою ветвь, что может означать значительную работу. Особенно трудная часть этой работы имеет дело с семантическими изменениями. Современные системы контроля версий могут творить чудеса с объединением сложных изменений в тексте программы, но они не понимают семантики кода. Если я изменил название функции, мой инструмент контроля версий может легко интегрировать внесенные мною изменения с изменениями Рейчел. Но если в своей ветви кода она добавит вызов функции, которая переименована в моей ветви, — код потерпит неудачу.

Проблема сложных слияний ухудшается с увеличением длины ветвей элементов экспоненциально. Интеграция четырехнедельных ветвей сложнее, чем двухнедельных, куда более чем в два раза. Поэтому многие люди настаивают на том, чтобы ветви были короткими — возможно, продолжительностью всего пару дней. Другие, такие как я, хотят, чтобы они были еще более короткими. Такой подход называется *непрерывной интеграцией* (Continuous Integration — CI), также известной как *магистральная разработка* (trunk-based development). При использовании CI каждый член команды интегрируется с магистральной линией по меньшей мере один раз в день. Это предотвращает слишком большое расхождение ветвей и, таким образом, значительно снижает сложность слияний. CI не является бесплатной: это означает, что вы используете методы, гарантирующие работоспособность магистральной линии, учитесь разбивать большие функциональные возможности на мелкие части и использовать флаги, чтобы отключать любые внутрипроцессные функциональные возможности, которые не могут находиться в нерабочем состоянии.

Поклонникам CI этот подход нравится отчасти потому, что он уменьшает сложность слияний, но главная причина предпочтения CI в том, что она гораздо более совместима с рефакторингом. Рефакторинг часто включает в себя внесение множества небольших изменений по всей кодовой базе, которые особенно

подвержены конфликтам семантического слияния (например, переименование широко используемой функции). Многие из нас встречались с командами, использующими ветви функциональных возможностей, которые находят рефакторинги настолько усугубляющими проблемы слияния, что прекращают заниматься рефакторингом полностью. CI и рефакторинг хорошо работают вместе, поэтому Кент Бек объединил их в *экстремальном программировании*.

Я не говорю, что вы не должны использовать ветви функциональных возможностей. Если они достаточно короткие, их проблемы значительно уменьшаются (пользователи CI обычно используют ветви, но интегрируют их с основной линией ежедневно). Функциональные ветви могут быть правильным подходом для проектов с открытым исходным кодом, где у вас есть нечастые фиксации кода от программистов, которых вы плохо знаете (и поэтому не слишком им доверяете). Но в команде разработчиков, занятых полный рабочий день, затраты, которые ветви накладывают на рефакторинг, чрезмерны. Даже если вы не переходите на полную CI, я все равно призываю вас к как можно более частой интеграции. Вам также следует учитывать объективные доказательства того, что команды, использующие CI, более эффективны в разработке программного обеспечения [10].

Тестирование

Одной из ключевых характеристик рефакторинга является то, что он не меняет наблюдаемое поведение программы. Если я тщательно выполняю рефакторинг, я ничего не сломаю. Но что, если (или, зная себя, — когда) я совершу ошибку? Ошибки случаются, но они не становятся проблемой, если они тут же выявляются. Каждый рефакторинг — это внесение небольшого изменения, значит, если я что-то и ломаю, то найти ошибку среди очень небольшого количества кода легко. Ну, а если я все же не могу ее обнаружить, то я могу вернуться к последней рабочей версии, почти ничего не потеряв.

Ключевым моментом является возможность быстрого обнаружения ошибки. Для этого мне нужно иметь возможность запуска комплексного набора тестов кода, причем работающего быстро, чтобы время его работы не удерживало меня от его частого запуска. Это означает, что в большинстве случаев, если я хочу выполнить рефакторинг, мне нужно иметь самотестируемый код [34].

Для некоторых читателей "самотестируемый код" звучит как нечто очень крутое, по сути, неосуществимое. Но за последние пару десятилетий я много раз видел, как команды создавали программное обеспечение именно таким образом. Это требует большого внимания и преданности тестированию, но преимущества такого подхода делают его действительно стоящим. Самотестируемый код позволяет не только выполнять рефакторинг, но и значительно безопаснее добавлять новые функциональные возможности, поскольку я могу быстро находить и

устранять любые обнаруженные ошибки. Ключевым моментом является то, что при сбое теста я могу просмотреть изменения, внесенные между тем, когда тесты в последний раз выполнялись правильно, и текущим кодом. При частых тестовых прогонах это будет всего лишь несколько строк кода. Точно зная, что именно эти несколько строк вызвали сбой, мне намного легче найти ошибку.

Это также ответ тем, кто полагает, что рефакторинг несет слишком большой риск появления ошибок. Без самотестируемого кода это разумное беспокойство, и именно поэтому я уделяю так много внимания надежным тестам.

Есть еще один способ справиться с проблемой тестирования. Если использовать среду с хорошими автоматизированными рефакторингами, то этим рефакторингам можно доверять даже без запуска тестов. Так можно выполнить составной рефакторинг при условии, что используются только те рефакторинги, которые безопасно автоматизированы. Это убирает из меню множество хороших рефакторингов, но все же оставляет их достаточно, чтобы получить от них некоторые преимущества. Я все равно предпочитаю иметь самотестируемый код, но в своем наборе инструментов полезно иметь и этот вариант.

Этот подход мотивирует стиль рефакторинга, использующий только ограниченный набор доказанно безопасных рефакторингов. Такие рефакторинги требуют аккуратного выполнения составных шагов и зависят от языка. Использующие их команды обнаружили, что они могут выполнять полезные рефакторинги для больших баз кода с плохим охватом тестами. Я не концентрируюсь на этой теме в данной книге, поскольку это более новая, менее описанная и понятная методика, которая включает в себя подробные, специфичные для языка действия. (Я надеюсь, в будущем расскажу об этом на своем веб-сайте. Чтобы понять, что это такое, — см. описание Джея Базузи (Jay Bazuzi) [3] более безопасного способа выполнения рефакторинга *Извлечение функции* (с. 152) в C ++.)

Неудивительно, что самотестируемый код тесно связан с непрерывной интеграцией — это механизм, который мы используем для выявления семантических конфликтов интеграции. Такие методы тестирования являются еще одним компонентом экстремального программирования и ключевой частью непрерывной поставки (continuous delivery).

Устаревший код

Большинство людей расценили бы большое наследство как нечто хорошее, но это один из тех случаев, когда взгляды программистов оказываются кардинально иными. Устаревший код обычно сложен, часто сопровождается плохими тестами, да еще и написан кем-то другим (невольная дрожь по телу).

Рефакторинг может быть фантастическим инструментом, помогающим понять устаревшую систему. Функции с вводящими в заблуждение именами можно

переименовывать, чтобы они имели смысл, неуклюжие программные конструкции сглаживаются, и программа превращается из неотесанного булыжника в отполированный камень. Но страшным драконом, охраняющим эту сокровищницу, обычно является недостаток тестов. Если у вас большая устаревшая система без тестов — вы не можете безопасно преобразовать ее в ясную и понятную.

Очевидный ответ на эту проблему заключается в добавлении тестов. Хотя это звучит очень просто (пусть и кропотливо), на практике все намного сложнее. Обычно систему легко тестировать, только если она была разработана с учетом требований к тестированию. Но в этом случае она будет иметь тесты, и беспокоиться не о чем.

Простого пути решения этой проблемы нет. Лучший совет, который я могу дать, — это купить книгу Майкла Физерса *Эффективная работа с унаследованным кодом* [7] и следовать ее указаниям. Не беспокойтесь о возрасте книги — ее советы так же верны, как и десятилетие назад. Грубо резюмируя, можно посоветовать вам протестировать систему, найдя в программе места, куда вы сможете вставить тесты. Создание таких мест включает в себя рефакторинг, который оказывается опаснее прочих, так как выполняется без тестирования, но это необходимый риск для продвижения вперед. В этой ситуации когда безопасный автоматический рефакторинг может оказаться удачной находкой. Если все это звучит сложно, то только потому, что это так и есть. К сожалению, никакого простого выхода из этой глубокой дыры нет — вот почему я такой сторонник написания самотестируемого кода изначально.

Даже если у меня есть все необходимые тесты, я не требую перестроить сложный унаследованный беспорядочный код в красивый. Что я реально предпочитаю делать — так это разбираться в необходимых частях кода. Каждый раз, разбираясь в некотором разделе кода, я пытаюсь сделать его немного лучше. Если это большая система, то больший рефакторинг будет проведен в тех областях, которые я посещаю чаще; — и это правильно, потому что если мне нужно часто работать с некоторым кодом, то, чем понятнее я его сделаю, тем большую отдачу получу.

Базы данных

Когда я писал первый вариант этой книги, я говорил, что рефакторинг баз данных является проблемной областью. Но уже в течение года после публикации книги ситуация изменилась. Мой коллега Прамод Садаладж (Pramod Sadalage) разработал подход к эволюционному проектированию баз данных [23] и рефакторингу баз данных [1], который широко используется в настоящее время. Суть этого метода заключается в объединении структурных изменений в схеме базы данных и коде доступа со сценариями переноса данных, которые можно легко создать для обработки больших изменений.

Рассмотрим простой пример переименования поля (столбца). Как и в рефакторинге *Изменение объявления функции* (с. 170), нужно найти исходное объявление структуры и все вызывающие ее объекты и изменить их в рамках одного изменения. Сложность, однако, заключается в том, что необходимо также преобразовать любые данные, использующие старое поле, таким образом, чтобы они использовали новое поле. Я пишу небольшой фрагмент кода, который выполняет это преобразование, и храню его в системе управления версиями вместе с кодом, который изменяет любую объявленную структуру и процедуры доступа. Затем, когда мне нужно выполнить миграцию между двумя версиями базы данных, я запускаю все существующие сценарии миграции между моей текущей копией базы данных и желаемой версией.

Как и при обычном рефакторинге, ключевой момент заключается в том, что каждое отдельное изменение оказывается небольшим, но фиксируется полное изменение, так что после миграции система продолжает работать. Поддержание малого размера изменений означает, что их легко написать, но их можно связать в последовательность, которая в состоянии внести существенные изменения в структуру базы данных и данные, хранящиеся в ней.

Одно из отличий от обычного рефакторинга состоит в том, что изменения базы данных часто лучше всего разделять на несколько производственных выпусков. Такой подход позволяет легко отменить любые изменения, которые вызывают проблемы в производственной системе. Таким образом, при переименовании поля первым изменением будет добавление нового поля базы данных без использования его. Затем можно настроить обновления таким образом, чтобы они обновляли одновременно как старые, так и новые поля. Затем постепенно читатели базы данных будут переводиться на новое поле. И только после того, как все они переместятся на новое поле, и работа в течение некоторого времени не обнаружит ошибок, можно будет удалить неиспользуемое старое поле. Такой подход к изменениям базы данных является примером общего подхода параллельных изменений [29].

Рефакторинг и архитектура

Рефакторинг глубоко изменил отношение людей к архитектуре программного обеспечения. В начале моей карьеры меня учили, что сначала следует тщательно проработать проект программного обеспечения и архитектуру, и они в основном должны быть завершены, прежде чем кто-то начнет писать код. Как только код был написан, его архитектура фиксировалась и могла быть изменена только из-за небрежности.

Рефакторинг отказывается от этой точки зрения и позволяет значительно менять архитектуру программного обеспечения, которое работало в течение многих лет. Рефакторинг может улучшить проект существующего кода, что подразумевается в подзаголовке данной книги. Но, как я указывал ранее, изменение унаследованного кода часто сопряжено с трудностями, в особенности если в нем отсутствуют приличные тесты.

Реальное влияние рефакторинга на архитектуру заключается в том, как он может быть использован для формирования хорошо спроектированной кодовой базы, способной элегантно реагировать на меняющиеся потребности. Самая большая проблема с завершенностью архитектуры перед началом кодирования заключается в том, что такой подход предполагает понимание и принятие требований к программному обеспечению на самой ранней стадии. Но опыт показывает, что это зачастую (и даже как правило) недостижимая цель. Неоднократно я видел, как люди начинают понимать, что им действительно нужно от программного обеспечения, только когда у них есть возможность его использовать, и видел влияние, которое этот факт оказывает на труд разработчиков.

Один из способов справиться с будущими изменениями — внедрить в программное обеспечение механизмы гибкости. Когда я пишу какую-то функцию, я вижу, что она имеет общую применимость. Чтобы обработать различные обстоятельства, в которых я могу ожидать ее использования, можно добавить к этой функции десяток параметров. Эти параметры и являются механизмами гибкости, и, как и большинство механизмов, оказываются далеко не бесплатными. Добавление параметров усложняет функцию для единственного применения, которое используется в настоящее время. Я обнаружил, что часто ошибаюсь в своих механизмах гибкости — либо потому, что меняющиеся потребности программы сработали не так, как я ожидал, либо конструкция моего механизма оказывается ошибочной. Если я учту все, что только можно, то в большинстве случаев мои механизмы гибкости фактически *замедлят* мою способность реагировать на изменения.

Используя рефакторинг, я могу прибегнуть к другой стратегии. Вместо того, чтобы рассуждать о том, какая гибкость мне понадобится в отдаленном будущем и какие механизмы позволят обеспечить ее наилучшим образом, я создаю программное обеспечение, которое удовлетворяет только текущие потребности, но делает это превосходно. Поскольку мое понимание потребностей пользователей со временем меняется, я использую рефакторинг для адаптации архитектуры программного обеспечения к этим новым требованиям. Я могу включить механизмы, которые не увеличивают сложность (например, небольшие, хорошо именованные функции), но любая гибкость, которая усложняет программное обеспечение, должна доказать свою необходимость, прежде чем будет включена в него. Если у меня нет различных аргументов для некоторого параметра при вызове функции

в настоящее время — я не буду вносить его в список параметров. Если придет время, когда мне нужно будет добавить этот параметр, я применю рефакторинг *Параметризация функции* (с. 355). Полезно также оценить, насколько сложно будет использовать рефакторинг для поддержки ожидаемых в будущем изменений. И только если я увижу, что позже рефакторинг окажется существенно сложнее, я подумаю о том, чтобы добавить механизм гибкости уже сейчас.

Этот подход к проектированию известен под разными именами: простое проектирование, инкрементное проектирование или yagni [42] (аббревиатура от "you aren't going to need it" — "вам это не понадобится"). Он не подразумевает отказ от архитектурного мышления, хотя иногда именно так наивно его и применяют. Я считаю, что yagni — это просто иной стиль включения архитектуры и проектирования в процесс разработки, стиль, который является бессмысленным без фундамента в виде рефакторинга.

Принятие yagni не означает, что я пренебрегаю архитектурным мышлением. Все еще есть случаи, когда выполнение рефакторинга — трудная задача, и небольшие предварительные размышления могут здорово сэкономить время. Но баланс сильно изменился — сейчас я гораздо более склонен решать проблемы позже, когда я лучше их понимаю.

Рефакторинг и процесс разработки программного обеспечения

Если вы читали предыдущий раздел о проблемах, то один из уроков, который вы, вероятно, извлекли, заключается в том, что эффективность рефакторинга связана с другими практиками разработки программного обеспечения, которые использует команда. Фактически раннее внедрение рефакторинга было частью экстремального программирования (Extreme Programming — XP) [40] — процесса, который был известен тем, что собрал воедино набор относительно необычных и взаимозависимых практик, таких как непрерывная интеграция, самотестируемый код и рефакторинг (последние два оказались вплетены в разработку, управляемую тестированием).

Экстремальное программирование было одним из первых методов гибкого (agile) программирования [27] и в течение нескольких лет привело к появлению гибкой методологии разработки. В настоящее время эти методы используются в достаточном количестве проектов, в которых гибкое мышление считается основным направлением (но в действительности большинство таких "гибких" проектов просто используют модное название). Чтобы действительно действовать гибко, команда должна быть не просто способна к рефакторингу, но и увлечена им — рефакторинг должен быть регулярной частью их работы.

Первым фундаментом рефакторинга является самотестируемый код. Под этим термином я подразумеваю наличие набора автоматических тестов, которые могу запускать в уверенности, что если в программе имеется ошибка, то какой-то из тестов обязательно провалится. Это настолько важная основа рефакторинга, что ей будет посвящена отдельная глава.

Для командного рефакторинга важно, чтобы каждый участник мог проводить рефакторинг в любое время, не мешая при этом работе других членов команды. Вот почему я поощряю непрерывную интеграцию. Благодаря ей усилия по рефакторингу каждого члена команды быстро передаются их коллегам. В результате никто не основывает новую работу на интерфейсах, которые удаляются другим, а если рефакторинг одного члена команды вызовет проблему у другого, то об этом станет тут же известно. Самотестируемый код является ключевым элементом непрерывной интеграции, и между тремя практиками — самотестируемым кодом, непрерывной интеграцией и рефакторингом — имеется тесная взаимосвязь.

С учетом этого трио практик мы используем подход к проектированию yagni, о котором я говорил в предыдущем разделе. Рефакторинг и yagni положительно укрепляют друг друга: не только рефакторинг (и его предпосылки) является основой для yagni, но и yagni облегчает проведение рефакторинга. Это связано с тем, что изменить простую систему легче, чем систему, в которой заложена большая гибкость. Сбалансируйте указанные практики, и вы сможете получить надежный код, который быстро реагирует на меняющиеся потребности.

Наличие этих фундаментальных практик дает основу для использования преимуществ других элементов гибкого подхода. Непрерывная поставка поддерживает наше программное обеспечение в состоянии постоянной готовности к выпуску. Это позволяет многим веб-организациям выпускать обновления много раз в день, но даже если нам это и не нужно, то это все равно снижает риски и позволяет нам планировать выпуски программного обеспечения не из-за технологических проблем, а для удовлетворения бизнес-потребностей. Имея прочную техническую базу, мы можем значительно сократить время, необходимое для претворения хорошей идеи в жизнь в производственном коде, что позволяет лучше обслуживать клиентов. Кроме того, эти методы повышают надежность программного обеспечения, уменьшая количество ошибок, требующих времени на их исправление.

Все это звучит довольно просто, но на практике это не совсем так. Разработка программного обеспечения, безотносительно используемого подхода, является сложным бизнесом, со сложным взаимодействием между людьми и машинами. Подход, который я описываю здесь, является проверенным способом справиться с этой сложностью, но, как и любой другой подход, он требует практики и навыков.

Рефакторинг и производительность

Распространенный вопрос, связанный с рефакторингом, — его влияние на производительность программы. Для облегчения понимания работы программы часто осуществляется модификация, приводящая в конечном итоге к замедлению программы. Это важный вопрос. Я не считаю возможным пренебрегать производительностью программного обеспечения, предпочитая чистоту проекта или надеясь на увеличение вычислительной мощности техники. Рефакторинг может приводить к замедлению программы, но при этом он делает более легкой настройку ее производительности. Секрет создания быстрых программ (если только они не предназначены для работы в режиме реального времени) состоит в том, чтобы начать с написания программы, производительность которой можно настраивать, а затем с помощью настройки достичь приемлемой скорости ее работы.

Я знаком с тремя подходами к написанию быстрых программ. Наиболее трудный путь зачастую применяется в системах с жесткими требованиями к выполнению в режиме реального времени. В этом случае при разложении проекта на составные части каждому компоненту выделяются определенные ресурсы времени и памяти. Компонент не должен выйти за рамки выделенного ресурса, хотя возможно применение механизма обмена ресурсами. При этом подходе жестко соблюдается выполнение программы в заданных временны*х* рамках. Это очень важно в таких системах, как, например, кардиостимуляторы, в которых опоздание может стоить человеческой жизни. В других системах, например в корпоративных информационных системах, с которыми я обычно имею дело, такой подход является избыточным.

Второй подход предполагает постоянное внимание к проекту. Каждый программист в любой момент времени делает все от него зависящее, чтобы обеспечивать высокую производительность программы. Этот подход широко распространен и кажется привлекательным, но на деле не так уж хорош. Вносимые изменения, повышающие производительность, обычно затрудняют работу с программой, что приводит к замедлению написания программы. Это могло бы быть разумной ценой, если бы в результате получалось более быстрое программное обеспечение, но, как правило, этого не происходит. Повышающие производительность изменения разбросаны по всей программе, и каждое из них относится только к некоторой узкой функциональности программы.

Нужно много времени, чтобы ничего не сделать

Программная система Chrysler Comprehensive Compensation работала слишком медленно. И пусть разработка еще не была завершена, это стало серьезно беспокоить разработчиков, потому что замедлялось выполнение тестов.

Кент Бек, Мартин Фаулер и я решили исправить такое положение дел. В ожидании встречи с коллегами я, будучи хорошо знакомым с системой, пытался понять, что же может так ее тормозить. Я подумал о нескольких причинах и поговорил с парнями о том, какие изменения могли бы потребоваться. Во время обсуждения нам пришло в голову несколько неплохих мыслей о том, как можно ускорить систему.

После этого мы измерили производительность с помощью профайлера. Увы, ни одна из предполагаемых мною причин не имела отношения к проблеме. Выяснилось, что система тратила половину рабочего времени на создание экземпляров класса дат. Самое интересное, что все эти объекты содержали одни и те же данные.

Изучив логику создания дат, мы придумали возможность оптимизировать этот процесс. При создании везде использовалось преобразование строк, даже если никакие внешние данные не вводились. Это было сделано просто для удобства ввода исходного текста, и это вполне можно было оптимизировать.

Затем мы рассмотрели применение этих дат. Выяснилось, что по большей части они участвовали в создании диапазонов дат, т.е. объектов, содержащих дату "от" и дату "до". Поразбиравшись еще немного, мы обнаружили, что большинство этих диапазонов пусты!

При работе с диапазонами дат было условлено, что любой диапазон, конечная дата которого предшествует начальной, является пустым. Это удобное соглашение, хорошо согласующееся с тем, как работает класс. При использовании данного соглашения вскоре выяснилось, что код, создающий диапазоны дат, начинающиеся после своего окончания, запутан и непонятен, так что мы вынесли эту функциональность в фабричный метод, создающий пустые диапазоны дат.

Это изменение должно было просто сделать код более ясным, но мы получили неожиданный бонус. Мы создали пустой диапазон в виде константы, которую фабричный метод возвращал вместо создания объекта каждый раз заново. После этой модификации скорость системы удвоилась, чего было вполне достаточно для выполнения тестов. Эта работа отняла у нас примерно пять минут.

Мы обсуждали вопрос, что может быть плохого в хорошо известном коде, со многими участниками проекта. Мы даже прикинули некоторые возможные усовершенствования кода, не проводя предварительных измерений.

> Оказалось, что мы полностью ошибались. Если не считать пары рассказанных во время обсуждения анекдотов, толку от него не было никакого.
>
> Вывод из этого рассказа следующий. Даже если вы точно знаете, как работает система, занимайтесь не гаданием, а замерами. Полученная информация в девяти случаях из десяти покажет, что ваши предположения были ошибочными!
>
> — *Рон Джеффрис (Ron Jeffries)*

Интересный факт, связанный с вопросами производительности, — в ходе анализа большинства программ выясняется, что бо́льшую часть времени работы программы выполняется небольшой фрагмент кода. Если оптимизировать весь код в одинаковой степени, то 90% оптимизации окажется выполненной впустую, так как это была оптимизация редко выполняемого кода. Время, затраченное на ускорение программы, и время, потерянное из-за ее невыразительности, потеряно при этом напрасно.

Третий подход к повышению производительности программы основан на упомянутой статистике. Он предполагает создание программы с высокой степенью декомпозиции на компоненты без учета получаемой производительности вплоть до этапа оптимизации последней (который обычно наступает на достаточно поздней стадии разработки и при выполнении которого осуществляется отдельная процедура настройки производительности программы).

Все начинается с запуска профайлера, который исследует выполнение программы и сообщает, где в основном расходуются время и память. Это позволяет выявить небольшой фрагмент кода, который представляет собой узкое место в смысле производительности. Усилия программистов сосредоточиваются в этих местах, и над ними осуществляется та же оптимизация, которая применяется при подходе с постоянным вниманием. Благодаря тому, что при этом усилия сосредоточены на узких местах, удается достичь бо́льших результатов при меньших усилиях. Но даже в такой ситуации необходима бдительность. Как и при рефакторинге, изменения следует вносить небольшими порциями, каждый раз компилируя, тестируя и выполняя профилирование. Если производительность не увеличивается, выполняется откат. Поиск и избавление от узких мест продолжается до достижения производительности, удовлетворяющей пользователей.

Разделение программы на компоненты способствует этому стилю оптимизации двумя путями. Во-первых, благодаря разделению появляется время, которое можно затратить на оптимизацию. В хорошо структурированный код быстрее добавляется новая функциональность и выигрывается время для работы над производительностью; профилирование же гарантирует, что это время не будет потрачено впустую. Во-вторых, хорошо структурированная программа обеспечивает

возможности более качественного анализа производительности. Профайлер указывает на узкие места в более мелких фрагментах кода, которые легче настроить. Благодаря более понятному коду легче выбрать возможные варианты настройки и понять, какая именно настройка может оказаться эффективной.

Я вижу, что рефакторинг позволяет мне быстрее писать программы. На какое-то время программы становятся более медленными, но зато их легче настраивать на этапе оптимизации. В конечном счете мы получаем больший выигрыш, чем без рефакторинга.

Истоки рефакторинга

Я не смог точно выяснить, когда именно появился термин *рефакторинг*. Хорошие программисты всегда посвящают хотя бы некоторое время наведению порядка в своем коде. Они занимаются этим, так как понимают, что проще модифицировать аккуратный код, а не сложный и запутанный, и хорошо известно, что написать такой хороший код сразу удается очень редко.

Рефакторинг идет дальше. В данной книге рефакторинг рассматривается как ведущий элемент процесса разработки программного обеспечения в целом. Первыми, кто понял важность рефакторинга, были Уорд Каннигем и Кент Бек, с 1980-х годов работавшие со Smalltalk. Smalltalk уже тогда был открытой для рефакторинга средой. Она очень динамична и позволяет быстро писать высокофункциональные программы. Smalltalk имеет короткий цикл "компиляция-компоновка-выполнение", обеспечивающий возможность быстро модифицировать программы. Этот язык программирования является объектно-ориентированным и предоставляет развитые средства для уменьшения влияния изменений, скрываемых за точно определенными интерфейсами. Уорд и Кент разработали методику создания программного обеспечения, приспособленную для применения в такой среде, и их работа вылилась в создание *экстремального программирования* (Extreme Programming). Они осознали значение рефакторинга для повышения производительности труда программистов и с тех пор применяют его в сложных программных проектах и совершенствуют данную технологию.

Идеи Уорда и Кента всегда оказывали сильное влияние на сообщество программистов на Smalltalk, так что понятие рефакторинга стало важным элементом культуры Smalltalk. Еще одна крупная фигура в сообществе программистов Smalltalk — Ральф Джонсон, профессор Университета штата Иллинойс в Урбана-Шампань, известный как член "банды четырех" [11]. Среди областей знания, в которых сосредоточены интересы Ральфа, находится создание программных каркасов (frameworks). Им также исследован вопрос о применении рефакторинга для создания эффективной и гибкой среды разработки.

Билл Опдайк (Bill Opdyke) был одним из аспирантов Ральфа и проявлял особый интерес к этим каркасам. Он обратил внимание на потенциальную ценность рефакторинга и заметил, что его применение не ограничивается рамками Smalltalk. Билл имел опыт разработки программного обеспечения для телефонных станций, характеризующегося нарастанием сложности со временем и трудностью модификации. Диссертация Билла рассматривала рефакторинг с точки зрения разработчика инструментальных средств. Билл изучил разновидности рефакторинга, которые могли оказаться полезными при разработке каркасов C++, и исследовал способы рефакторинга, сохраняющие семантику, а также способы доказательства сохранения семантики и возможности реализации своих идей в инструментальных средствах. Диссертация Билла [44] является на сегодняшний день одной из самых влиятельных работ на тему рефакторинга.

Я встречался с Биллом на конференции OOPSLA в 1992 году. За чашкой кофе мы обсуждали некоторые стороны моей работы по созданию концептуальных каркасов в здравоохранении. Билл рассказал мне о своих исследованиях, и, помню, я тогда подумал, что это интересно, но не имеет практического применения. Как же я ошибался!

Джон Брант (John Brant) и Дон Робертс значительно продвинули идеи инструментальных средств рефакторинга, создав инструмент для выполнения рефакторинга в Smalltalk — Refactoring Browser.

Ну, а что же я? Мне всегда нравился хороший код, но я никогда не думал, что это так важно. Затем я участвовал в одном проекте с Кентом и увидел, как он применяет рефакторинг. Я понял, какое огромное влияние он оказывает на производительность труда и качество результатов. Этот опыт убедил меня в том, что рефакторинг — очень важная технология разработки программного обеспечения. Но я был разочарован отсутствием книг, которые мог бы прочесть практикующий программист, а никто из упомянутых выше экспертов по этому вопросу и не собирался писать такую книгу! Мне пришлось брать этот труд на себя — с их помощью.

К счастью, концепция рефакторинга завоевала популярность в программной индустрии. Книга хорошо продавалась, и слово "рефакторинг" прочно вошло в лексикон большинства программистов. Появилось больше инструментов, особенно для Java. Одним из недостатков этой популярности было слишком вольное использование термина *рефакторинг* для обозначения любого вида реструктуризации.

Автоматизированные рефакторинги

Возможно, самое большое изменение в рефакторинге за последнее десятилетие — это появление инструментов, поддерживающих автоматический рефакторинг. Если я хочу переименовать метод в Java и использую IntelliJ IDEA [13] или,

скажем, Eclipse [6], то я могу сделать это, просто выбрав соответствующий пункт меню. Инструмент выполняет рефакторинг вместо меня, и я обычно достаточно уверен в его корректности, чтобы даже не потрудиться запустить набор тестов.

Первым инструментом, который оказался способен на это, был браузер Smalltalk Refactoring, написанный Джоном Брантом и Доном Робертсом. В начале века эта идея получила широкое распространение в сообществе Java. Когда JetBrains выпустил свою IDE IntelliJ IDEA, одной из ее неотъемлемых функций стал автоматический рефакторинг. Вскоре после этого IBM последовала их примеру с инструментами рефакторинга в Visual Age for Java. Visual Age не оказал большого влияния, но многие его возможности были реализованы в Eclipse, включая поддержку рефакторинга.

Рефакторинг также пришел и в C#, первоначально через Resharper JetBrains, плагин для Visual Studio. Позже команда Visual Studio добавила в среду некоторые возможности рефакторинга.

В настоящее время довольно часто встречается та или иная поддержка рефакторинга в редакторах и инструментах, хотя реальные возможности такой поддержки значительно варьируются. Одни изменения связаны с конкретным инструментарием, другие — с ограничениями возможностей автоматического рефакторинга на разных языках. Я не собираюсь анализировать здесь возможности различных инструментов, но думаю, что имеет смысл немного поговорить о некоторых основополагающих принципах.

Грубый способ автоматизировать рефакторинг — использовать манипуляции с текстом, такие как поиск/замена для изменения имени, или простая реорганизация кода в рефакторинге *Извлечение переменной* (с. 165). Это очень грубый подход, которому, безусловно, нельзя доверять без последующего тестирования. Но это может быть удобным первым шагом. Я использую такие макросы в Emacs для ускорения работы по рефакторингу, когда у меня нет более сложных доступных рефакторингов.

Чтобы правильно выполнить рефакторинг, инструмент должен работать не с текстом кода, а с его синтаксическим деревом. Управление синтаксическим деревом намного надежнее для сохранения поведения кода. Вот почему в настоящее время большинство возможностей рефакторинга являются частью мощных интегрированных сред разработки, которые используют синтаксическое дерево не только для рефакторинга, но также для навигации по коду и множества других функциональных возможностей. Сотрудничество между текстом и синтаксическим деревом выводит возможности этих интегрированных сред разработки далеко за пределы простых текстовых редакторов.

Рефакторинг — это не просто понимание и обновление синтаксического дерева. Инструмент должен также выяснить, как преобразовать код в текст в окне редактора. Словом, реализация достойного автоматического рефакторинга — сложная

программистская задача, о всех тонкостях которой я, как правило, и не подозреваю, когда с удовольствием использую соответствующие инструменты.

Многие рефакторинги становятся намного безопаснее при применении к языку со статической типизацией. Рассмотрим простой рефакторинг *Изменение объявления функции* (с. 170). У меня могут быть методы `addClient` в классах `Salesman` и `Server`. Я хочу переименовать метод в классе `Salesman`. По своему предназначению он отличается от одноименного метода `Server`, переименовывать который я не хочу. Без статической типизации инструменту будет трудно определить, связан ли какой-то из вызовов `addClient` с `Salesman`. Браузер рефакторинга генерирует список точек вызовов, и я сам решаю, какие из них следует изменить. Это делает данный рефакторинг небезопасным, что заставляет меня проводить тестирование. Такой инструмент все же полезен, но эквивалентная операция в Java может быть полностью безопасной и автоматизированной. Поскольку инструмент может выявить правильный класс метода с помощью статической типизации, я могу быть уверен, что будет изменено название только тех методов, для которых это нужно сделать.

Инструменты часто идут дальше. Если я переименовываю переменную, мне могут предложить внести изменения в комментарии, использующие это имя. Если я выполняю рефакторинг *Извлечение функции* (с. 152), инструмент обнаруживает код, который дублирует тело новой функции, и предлагает заменить его вызовом. Программирование с помощью таких мощных рефакторингов является веской причиной использовать IDE, а не придерживаться привычного текстового редактора. Лично я большой любитель Emacs, но при работе в Java я предпочитаю IntelliJ IDEA или Eclipse — в значительной степени из-за их поддержки рефакторинга.

Хотя сложные инструменты рефакторинга выглядят почти волшебными в своей способности безопасно выполнять рефакторинг кода, есть некоторые исключительные случаи, когда они могут не сработать. Например, менее зрелые инструменты страдают от рефлексивных вызовов, таких как `Method.invoke` в Java (хотя более зрелые инструменты справляются с этим довольно хорошо). Таким образом, время от времени целесообразно запускать тестовый набор даже при использовании в основном безопасных рефакторингов, чтобы убедиться, что все в порядке. Обычно я выполняю рефакторинг с использованием некоторой комбинации автоматического и ручного рефакторинга, а потому запускаю тесты достаточно часто.

Возможность использования синтаксического дерева для анализа и рефакторинга программ является неоспоримым преимуществом интегрированной среды разработки по сравнению с простым текстовым редактором. Тем не менее многие программисты предпочитают гибкость своего любимого текстового редактора и хотели бы иметь и то, и другое. Технологией, в настоящее время набирающей

обороты, является Language Servers [15]: программное обеспечение, которое формирует синтаксическое дерево и предоставляет API для текстовых редакторов. Такие языковые серверы могут поддерживать множество текстовых редакторов и предоставлять команды для выполнения сложного анализа кода и операций рефакторинга.

Что дальше

Кажется немного странным говорить о дальнейшем чтении уже во второй главе, но это место ничуть не хуже другого, чтобы указать, что имеется множество материалов по рефакторингу, которые выходят за рамки этой книги.

Данная книга научила рефакторингу многих программистов, но я в большей степени пытаюсь сделать справочник, а не учебник по рефакторингу. Если вы ищете учебник, я бы предложил книгу Билла Уэйка (Bill Wake) *Refactoring Workbook* [47], которая содержит множество практических упражнений по рефакторингу.

Многие из тех, кто впервые применил рефакторинг, проявляют активность и в сообществе проектных шаблонов. Джошуа Кериевски (Josh Kerievsky) тесно связывает эти два мира в книге *Рефакторинг с использованием шаблонов* [14], рассматривая наиболее ценные проектные шаблоны из великолепной книги "банды четырех" [11], и показывает, как использовать рефакторинг для эволюции в указанном направлении.

Данная книга концентрируется на рефакторинге в программировании общего назначения, но рефакторинг применим и в специализированных областях. Особого внимания в этом плане заслуживают книги *Рефакторинг баз данных* [1] Скотта Амблера (Scott Ambler) и Прамода Садаладжа (Pramod Sadalage) и *Refactoring HTML* [12] Эллиота Расти Гарольда (Elliotte Rusty Harold).

Хотя в названии следующей книги нет слова "рефакторинг", ее все равно стоит упомянуть — это книга Майкла Фезерса (Michael Feathers) *Эффективная работа с унаследованным кодом*, которая в первую очередь представляет собой книгу о том, как выполнять рефакторинг старой кодовой базы с плохим охватом тестами.

Хотя эта книга (и ее предшественница) предназначены для программистов на любом языке программирования, есть место для книг по рефакторингу для конкретных языков. Двое моих бывших коллег, Джей Филдс (Jay Fields) и Шейн Харви (Shane Harvey), написали такую книгу для языка программирования Ruby [8].

Для получения более свежих материалов обратитесь к веб-странице данной книги, а также к основному веб-сайту по рефакторингу: `refactoring.com` [46].

Глава 3

Запахи в коде

Авторы: Кент Бек и Мартин Фаулер

Если что-то стало **пованивать**, *его лучше сменить.*
— Бабушка Бек при обсуждении проблем подгузников

Сейчас вы уже должны хорошо представлять себе, как действует рефакторинг. Но если вы знаете "как", это еще не значит, что вы знаете "когда". Решение о том, когда приступать к рефакторингу и когда прекратить его выполнение, не менее важно, чем умение выполнять рефакторинг.

И здесь мы сталкиваемся с дилеммой. Легко объяснить, как удалить переменную экземпляра или создать иерархию, — это достаточно простые вопросы, в отличие от объяснения, когда это следует делать. Вместо того чтобы взывать к расплывчатым представлениям об эстетике программирования (честно говоря, мы, консультанты, часто поступаем именно так), я попытался подвести под это более прочную основу.

Я как раз размышлял над этим сложным вопросом, когда посетил Кента Бека в Цюрихе. Возможно, он тогда находился под впечатлением запахов от своей новорожденной дочки, потому что выразил представление о том, когда проводить рефакторинг, именно с использованием этой аналогии.

Вы можете спросить: "Чем, собственно, запах лучше туманных рассуждений об эстетике?" Но мне кажется, что использование понятия запаха лучше рассуждений об эстетичности. Мы просмотрели очень большое количество кода, написанного для разных проектов, степень успешности которых простиралась в очень широком диапазоне — от весьма удачных до полудохлых. При этом мы научились искать в коде определенные признаки, которые предполагают возможность рефакторинга (а иногда просто кричат о его необходимости). (Слово "мы" в этой главе отражает тот факт, что у главы два автора — я и Кент. Определить, кто и что писал, просто: авторство смешных шуток принадлежит мне, а всего остального — ему.)

Мы не будем даже пытаться дать точные критерии необходимости рефакторинга. Наш опыт показывает, что никакие системы показателей не смогут соперничать с человеческой интуицией, основанной на знаниях. Мы приведем лишь симптомы неприятностей, устраняемых путем рефакторинга. У вас должно

развиться собственное чутье, какое количество следует считать "чрезмерным" для атрибутов класса или строк кода для метода.

Эта глава и таблица в конце книги должны подсказать, когда вам неясно, какие именно методы рефакторинга применять. Прочтите эту главу (или просмотрите таблицу) и попробуйте выяснить, какой именно запах вы чувствуете. Затем обратитесь к предлагаемым методам рефакторинга и проверьте, подойдут ли они вам. Вполне возможно, что запах не идентичен, но общее направление вполне может быть выбрано правильно.

Таинственное имя

Размышления над книгой в попытках понять, что же происходит, — это замечательно, когда вы читаете детективный роман, а не код. Мы можем мечтать о том, чтобы быть агентом под прикрытием, но наш код должен быть простым и понятным. Одна из наиболее важных частей ясного кода — это хорошие имена, поэтому мы много думаем об именовании функций, модулей, переменных, классов, чтобы они четко сообщали, что делают и как их использовать.

К сожалению, именование — одна из двух трудностей в программировании [38]. Таким образом, возможно, наиболее распространенными рефакторингами, которые мы выполняем, являются переименования: *Изменение объявления функции* (с. 170) (для переименования функции), *Переименование переменной* (с. 183) и *Переименование поля* (с. 289). Люди часто боятся переименовывать вещи, думая, что это не стоит затрачиваемого труда, но на самом деле хорошее имя может сэкономить часы озадаченного непонимания в будущем.

Переименование — это не просто упражнение по смене имен. Когда вы не можете придумать хорошее название для чего-то, это часто является признаком более глубокого проектного недуга. Недоумение по поводу хитрого названия часто в конечном итоге ведет к значительным упрощениям нашего кода.

Дублируемый код

Увидев одинаковые структуры кода в нескольких местах, можно быть уверенным, что если их удастся объединить, то программа от этого только выиграет. Дублирование означает, что каждый раз, когда вы читаете такие копии, вам необходимо внимательно проверять — нет ли между ними какой-либо разницы. Если вам нужно изменить дублированный код, вы должны найти и исправить каждую копию.

Простейшая задача с дублированием кода возникает, когда одно и то же выражение присутствует в двух методах одного и того же класса. В этом случае достаточно применить рефакторинг *Извлечение функции* (с. 152) и вызывать код вновь созданного метода из обеих точек. Если код похож, но полностью не совпадает, можно применить рефакторинг *Перемещение инструкций* (с. 269), чтобы совпадающие фрагменты были собраны вместе для облегчения их выделения. Если дублируемый код находится в подклассах одного базового класса, можно применить рефакторинг *Подъем метода* (с. 394), чтобы избежать вызова одного из другого.

Длинная функция

По нашему опыту, программы, использующие короткие функции, живут долго и счастливо. Программистам с малым опытом работы часто кажется, что на самом деле никаких вычислений не происходит, а программы состоят только из нескончаемой цепочки делегирований действий от одного объекта к другому. Однако, работая с такой программой на протяжении нескольких лет, вы понимаете, какую ценность представляют собой маленькие функции. Все выгоды, которые дает косвенность — понятность, совместное использование и выбор, — поддерживаются именно маленькими функциями.

Еще на заре программирования было ясно, что чем длиннее функция, тем труднее понять, как она работает. В старых языках программирования вызов процедур был связан с большими накладными расходами, которые удерживали программистов от применения маленьких функций. Современные объектно-ориентированные языки в значительной мере устранили накладные расходы вызовов. Но издержки сохраняются для того, кто читает код, так как ему приходится переключаться между контекстами, чтобы понять, что делает та или иная процедура. Среда разработки, позволяющая видеть одновременно две функции, помогает облегчить это переключение; но главное, что способствует пониманию маленьких функций, — это присвоение им разумных имен. Правильно выбранное имя функции зачастую позволяет не изучать ее тело.

В итоге мы приходим к необходимости активнее применять декомпозицию функций. Эвристическое правило, которому мы следуем, гласит, что если возникает необходимость что-то прокомментировать, то пора писать функцию. В этой функции содержится код, который требовал комментариев, но ее название отражает его *назначение*, а не способ решения им задачи. Такая процедура может применяться к группе строк или даже к единственной строке кода. К ней можно прибегнуть даже тогда, когда вызов функции оказывается длиннее, чем замененный ею код, — при условии, что имя метода разъясняет его предназначение.

Главным является не длина метода, а семантическое расстояние между тем, что метод делает, и тем, как он это делает.

В 99% случаев, чтобы укоротить метод, требуется использовать рефакторинг *Извлечение функции* (с. 152). Найдите части функции, которые кажутся связанными между собой, и образуйте новую функцию.

Если у вас есть функция со множеством параметров и временных переменных, это мешает выделению новой функции. Применяя рефакторинг *Извлечение функции* (с. 152), приходится передавать такое количество параметров и временных переменных, что результат оказывается ничуть не проще, чем оригинал. Устранить временные переменные зачастую можно с помощью рефакторинга *Замена временной переменной запросом* (с. 225), а длинные списки параметров можно сократить с помощью рефакторингов *Введение объекта параметра* (с. 186) и *Сохранение всего объекта* (с. 364).

Если после этого все равно остается слишком много временных переменных и параметров, приходится вызывать тяжелую артиллерию, а именно — рефакторинг *Замена функции командой* (с. 381).

Как же выявить фрагменты кода, которые должны быть выделены в отдельные функции? Хороший способ — поискать комментарии: они часто указывают на такое семантическое расстояние. Блок кода с комментариями говорит о том, что его можно заменить функцией, имя которой основано на этом комментарии. Даже одна строка может быть выделена в функцию, если она нуждается в пояснениях.

Условные инструкции и циклы также служат признаками возможного выделения в метод. Для работы с условными выражениями подходит рефакторинг *Декомпозиция условной инструкции* (с. 306). Большая инструкция `switch` должна быть превращена в одиночные вызовы функций с помощью рефакторинга *Извлечение функции* (с. 152). Если имеется более одной инструкции `switch`, включающей одно и то же условие, следует применить рефакторинг *Замена условной инструкции полиморфизмом* (с. 317).

В случае цикла извлеките его и содержащийся в нем код в отдельную функцию. Если вам трудно присвоить имя извлеченному циклу, это может быть связано с тем, что он решает две разные задачи, и в этом случае не бойтесь использовать рефакторинг *Разделение цикла* (с. 274) для их разделения.

Длинный список параметров

Ранее при обучении программированию все необходимые подпрограмме данные рекомендовалось передавать в виде параметров. Это можно понять, потому что альтернативой были глобальные переменные, а глобальные переменные — это одна большая головная боль. Но в длинных списках параметров трудно ра-

зобраться. Они зачастую противоречивы и сложны в применении, а кроме того, их приходится вечно изменять по мере возникновения необходимости в новых данных.

Если можно получить параметр с помощью запроса другого параметра, можно использовать рефакторинг *Замена параметра запросом* (с. 369), чтобы удалить один из параметров. Рефакторинг *Сохранение всего объекта* (с. 364) позволяет заменить набор данных, получаемых от существующего объекта, самим этим объектом. Если же имеется ряд элементов данных без логического объекта, их можно скомбинировать в единое целое с помощью рефакторинга *Введение объекта параметра* (с. 186). Если параметр используется в качестве флага для диспетчеризации разного поведения, попробуйте применить рефакторинг *Удаление аргумента-флага* (с. 359).

Отличным способом уменьшить размеры списка параметров являются классы. Они особенно полезны, когда несколько функций совместно используют несколько значений параметров. В таком случае вы можете использовать рефакторинг *Объединение функций в класс* (с. 190) для представления этих общих значений в виде полей. Если бы мы надели наши шляпы функционального программирования, то могли бы сказать, что это создает набор частично применяемых функций.

Глобальные данные

С самых первых дней участия в разработке программного обеспечения нас предупреждали об опасностях глобальных данных, и о том, как они были изобретены демонами из четвертого уровня ада, который является местом отдыха любого программиста, который осмеливается им пользоваться. И хотя мы несколько скептически относимся к огню и сере, это по-прежнему один из наиболее острых запахов, с которыми мы можем столкнуться. Проблема глобальных данных заключается в том, что они могут быть изменены из любого места кодовой базы, и нет никакого механизма, чтобы определить, какой именно фрагмент их изменил. Это дальнодействие приводит к ошибкам, источник которых крайне трудно выявить. Наиболее очевидной формой глобальных данных являются глобальные переменные, но та же проблема возникает и при использовании переменных класса и синглтонов.

Ключевой защитой является рефакторинг *Инкапсуляция переменной* (с. 178), который всегда является нашим первым шагом при встрече с данными, открытыми для изменения любой частью программы. По крайней мере, когда вы завернете их в функцию, то начнете видеть, какой код их изменяет, и контролировать обращение к ним. Далее полезно максимально ограничить область видимости функции, перемещая ее в класс или модуль, где ее сможет увидеть только код этого модуля.

Глобальные данные особенно неприятны, когда они изменяются. Если некоторые глобальные данные гарантированно не меняются после запуска программы, они относительно безопасны — если, конечно, ваш язык программирования может обеспечить такую гарантию.

Глобальные данные иллюстрируют максиму Парацельса: разница между ядом и лекарством — в дозировке. Вы можете удалить небольшие дозы глобальных данных, но с ростом их количества справляться с ними становится экспоненциально сложнее. Даже небольшое их количество следует инкапсулировать, чтобы по мере развития программного обеспечения быть в состоянии справляться с его изменениями.

Изменяемые данные

Изменения в данных часто могут вести к неожиданным последствиям и хитрым ошибкам. Я могу обновить некоторые данные в одном месте, не зная, что другая часть программного обеспечения ожидает какие-то другие данные и теперь терпит неудачу. Такой сбой особенно трудно обнаружить, если он происходит только в редких условиях. По этой причине целая школа разработки программного обеспечения (функциональное программирование) основана на идее, что данные никогда не должны изменяться, и что обновление структуры данных всегда должно возвращать новую копию структуры с изменениями, оставляя старые данные нетронутыми.

Эти языки, однако, все еще являются относительно небольшой частью мира программирования; многие из нас работают на языках, которые позволяют переменным изменяться. Но это не означает, что мы должны игнорировать преимущества неизменяемости — мы можем многое сделать, чтобы ограничить риски неограниченного обновления данных.

Можно использовать рефакторинг *Инкапсуляция переменной* (с. 178), чтобы гарантировать, что все обновления данных происходят через узкие рамки функций, которые легче отслеживать и развивать. Если переменная обновляется, чтобы хранить совершенно иную информацию, рефакторинг *Расщепление переменной* (с. 285) разделит хранимую информацию и позволит избежать рискованного обновления. Постарайтесь вынести как можно больше логики из кода, который выполняет обновление, используя рефакторинги *Перемещение инструкций* (с. 269) и *Извлечение функции* (с. 152), чтобы отделить код без побочных эффектов от кода, который выполняет обновление. В различных API используйте рефакторинг *Отделение запроса от модификатора* (с. 352), чтобы гарантировать, что вызывающим функциям не нужно вызывать код, который имеет побочные действия, если только в этом нет реальной необходимости. Мы предпочитаем также использовать

рефакторинг *Удаление метода установки значения* (с. 376) везде, где только можем, так как иногда просто попытка найти клиентов функции установки значения помогает найти возможности уменьшения области видимости переменной.

Изменяемые данные, которые могут быть вычислены в другом месте, являются особенно неприятными. Это не просто богатый источник путаницы, ошибок и пропущенных домашних ужинов — они просто не нужны. Нужно опрыскать их концентрированным раствором уксуса и выполнить рефакторинг *Замена вычисленной переменной запросом* (с. 293).

Изменяемые данные не являются большой проблемой, когда они представляют собой переменную, область видимости которой состоит всего из пары строк, но риск возрастает с ростом области видимости. Используйте рефакторинги *Объединение функций в класс* (с. 190) или *Объединение функций в преобразование* (с. 195), чтобы ограничить объем кода, которому необходимо обновление переменной. Если переменная содержит некоторые данные со внутренней структурой, обычно лучше не модифицировать ее на месте, а заменить всю структуру, используя рефакторинг *Замена ссылки значением* (с. 296).

Расходящиеся изменения

Мы структурируем программы, чтобы облегчить внесение в них изменений; в конце концов, программное обеспечение тем и отличается от аппаратного обеспечения, что его можно изменять. Планируя изменение, мы хотим иметь возможность перейти в определенную точку программы и внести изменения именно в ней. Если сделать это не удается, то здесь пахнет сразу двумя тесно связанными проблемами.

Расходящиеся (divergent) изменения имеют место тогда, когда один модуль часто изменяется различными путями по разным причинам. Если, глядя на модуль, вы замечаете, что вот эти три функции придется изменять для каждой новой базы данных, а вот эти четыре функции — при каждом появлении нового финансового требования, то это является признаком расходящихся изменений. Взаимодействие с базой данных и обработка финансов имеют разные контексты, так что можно облегчить свою программистскую жизнь, перемещая эти контексты в разные модули. Таким образом, когда требуется изменение в одном контексте, следует разобраться только с ним одним и игнорировать другой. Мы всегда считали это важным, но теперь, когда наш мозг с возрастом уменьшается в размерах, этот подход становится просто необходимым. Конечно, такая ситуация часто обнаруживается только после добавления нескольких баз данных или финансовых инструментов; обычно в начале существования программы границы контекстов неясны и продолжают перемещаться по мере эволюции разрабатываемого программного обеспечения.

Если два аспекта естественным образом образуют последовательность (например, вы получаете данные из базы данных, а затем применяете к ним финансовую обработку), то рефакторинг *Разделение этапа* (с. 201) разделяет их с точной структурой данных между ними. Если среди вызовов много вызовов в одном и другом направлении — создайте соответствующие модули и используйте рефакторинг *Перенос функции* (с. 244), чтобы разделить обработку по модулям. Если два типа обработки смешаны прямо в функциях, используйте рефакторинг *Извлечение функции* (с. 152) для разделения обработки перед перемещением функций. Если модули являются классами, то рефакторинг *Извлечение класса* (с. 229) помогает формализовать выполнение разбиения.

Стрельба дробью

"Стрельба дробью" подобна расходящимся изменениям, но представляет собой их противоположность. Унюхать ее можно, когда при выполнении любых изменений приходится вносить множество мелких модификаций в большое число различных модулей. Когда изменения разбросаны повсюду, их трудно искать и можно пропустить важное изменение.

В этой ситуации следует свести все изменения в один модуль, использовав рефакторинги *Перенос функции* (с. 244) и *Перенос поля* (с. 253). Если у вас есть целое множество функций, работающих с похожими данными, используйте рефакторинг *Объединение функций в класс* (с. 190). Если есть функции, которые преобразуют или обогащают структуру данных, используйте *Объединение функций в преобразование* (с. 195). Часто оказывается полезен рефакторинг *Разделение этапа* (с. 201), если общие функции могут объединять свои выходные данные для потребления следующей фазой логики.

Полезная тактика в случае стрельбы дробью заключается в использовании таких рефакторингов, как *Встраивание функции* (с. 161) или *Встраивание класса* (с. 232) для объединения плохо разделенной логики. Вы получите запах в виде длинной функции или большого класса, но сможете прибегнуть к другим рефакторингам для более разумного разбиения логики на части. Несмотря на нашу чрезмерную любовь к маленьким функциям и классам, мы не боимся создавать в качестве промежуточного шага реорганизации что-то большое.

Завистливые функции

Когда мы делим программы на модули, мы пытаемся разделить код на такие зоны, чтобы максимизировать взаимодействие внутри зоны и минимизировать

взаимодействие между зонами. Классический пример — когда функция в одном модуле тратит больше времени на общение с функциями или данными внутри другого модуля, чем в своем собственном. Мы потеряли счет тому, сколько раз видели функции, вызывающие десяток функций доступа к данным другого объекта для вычисления какого-то значения. К счастью, лекарство очевидно: функция четко хочет быть поближе к данным, поэтому используйте рефакторинг *Перенос функции* (с. 244), чтобы свести их вместе. Иногда этим недугом страдает только часть функции, и в этом случае используйте рефакторинг *Извлечение функции* (с. 152) для данной части, а затем с помощью рефакторинга *Перенос функции* (с. 244) дайте функции собственный дом.

Конечно, не все ситуации так очевидны. Часто функция использует функции сразу из нескольких модулей — так в какой из них ее следует поместить? Мы используем эвристику, заключающуюся в определении того, в каком классе находится больше всего данных, и перемещении функции к этим данным. Этот шаг зачастую оказывается легче, если использовать рефакторинг *Извлечение функции* (с. 152), чтобы разбить функцию на части, которые будут размешены в разных местах.

Конечно, могут быть сложные схемы, нарушающие это правило. На ум сразу приходят проектные шаблоны "Стратегия" и "Визитер" [11]. Еще одним примером является проектный шаблон "Самоделегирование" [4]. С его помощью можно бороться с запахом расходящихся изменений. Фундаментальное практическое правило гласит: то, что изменяется одновременно, лучше хранить в одном месте. Данные и функции, использующие эти данные, как правило, изменяются вместе, но бывают и исключения. Сталкиваясь с ними, мы перемещаем поведение так, чтобы изменения осуществлялись в одном месте. Проектные шаблоны "Стратегия" и "Визитер" позволяют легко изменять поведение, потому что они изолируют небольшое количество поведения, которое должно быть перекрыто, ценой увеличения косвенности.

Группы данных

Данные — как дети: они любят сбиваться в тесные группы. Часто можно видеть, как одни и те же три-четыре элемента данных встречаются во множестве мест: поля в паре классов, параметры в нескольких методах. Данные, встречающиеся совместно, имеет смысл превращать в отдельный класс. Сначала следует найти, где группы данных встречаются в качестве полей, и, применяя к ним рефакторинг *Извлечение класса* (с. 229), преобразовать группы данных в объект. Затем следует обратить внимание на сигнатуры методов и применить рефакторинг *Введение объекта параметра* (с. 186) или *Сохранение всего объекта* (с. 364) для

сокращения их объема. Непосредственной выгодой от этого являются сокращение многих списков параметров и упрощение вызовов методов. Не беспокойтесь о том, что некоторые группы данных используют лишь часть полей нового объекта. Заменив несколько полей новым объектом, вы оказываетесь в выигрыше.

Хорошая проверка заключается в том, чтобы удалить одно из значений данных и посмотреть, сохранят ли при этом смысл остальные данные. Если нет, то это верный признак того, что данные лучше объединить в один объект.

Заметьте, что мы выступаем за создание класса, а не простой структуры записей. Мы делаем это, потому что использование класса позволяет добиться приятного запаха. Теперь вы можете искать случаи завистливых функций, что приведет вас к поведению, которое может быть перемещено в новые классы, еще до того, как они станут полезными членами программы.

Одержимость примитивами

Большинство сред программирования основаны на широком применении набора примитивных типов: целых чисел, чисел с плавающей точкой и строк. Библиотеки могут добавить такие дополнительные небольшие объекты, как даты. Встречается немало программистов, которые не хотят создавать свои собственные фундаментальные типы, полезные для их предметной области, — например денежные единицы, координаты или диапазоны. И потому приходится встречаться с расчетами, которые рассматривают денежные суммы как простые числа; с вычислениями физических величин, которые игнорируют единицы измерений и добавляют дюймы к миллиметрам; или с большим количеством кода наподобие `if(a<upper && a>lower)...`.

Строки являются особенно яркими источниками запаха этого вида: номер телефона — это нечто большее, чем просто набор символов. Если даже такой тип не делает ничего иного — он по крайней мере включает согласованную логику отображения в пользовательском интерфейсе. Представление таких типов в виде строк является настолько распространенным запахом, что некоторые называют их "строково типизированными" переменными.

Выбраться обратно в мир значимых типов помогает рефакторинг *Замена примитива объектом* (с. 221). Если значение примитивного данного представляет собой код типа для управления условным поведением, воспользуйтесь рефакторингом *Замена кода типа подклассами* (с. 405) с последующим рефакторингом *Замена условной инструкции полиморфизмом* (с. 317).

При наличии группы полей, которые должны работать совместно, применяйте рефакторинги *Извлечение класса* (с. 229) и *Введение объекта параметра* (с. 186).

Повторяющиеся switch

Поговорите с настоящими поклонниками объектно-ориентированного программирования, и они очень быстро объяснят вам всю порочность инструкций `switch`. Они будут утверждать, что любая инструкция `switch`, которую вы видите, просит применить рефакторинг *Замена условной инструкции полиморфизмом* (с. 317). Мы даже слышали от некоторых, что вся условная логика должна быть заменена полиморфизмом, а большинство инструкций `if` следует выбросить в мусорное ведро истории.

Даже будучи юношами с горящими глазами мы никогда не были безоговорочно настроены против условных инструкций. Да, в первом издании этой книги был описан запах, озаглавленный "Инструкции `switch`". Он появился потому, что в конце 1990-х годов мы обнаружили, что полиморфизм, к сожалению, недооценивается, и видели выгоду в том, чтобы заставить людей активнее его использовать.

В наши дни полиморфизм используется куда активнее, так что теперь инструкция `switch` — вовсе не красный флаг, на который надо бросаться, как пятнадцать лет назад. Кроме того, многие языки поддерживают более сложные версии инструкций `switch`, которые используют в качестве своей базы не просто некоторый примитивный числовой код. Так что сейчас мы сосредоточимся на повторении `switch`, когда одна и та же условная логика (либо в `switch`/`case`, либо в каскаде `if`/`else`) появляется в разных местах. Проблема таких дублирующихся `switch` состоит в том, что всякий раз, когда вы добавляете ветвь, вы должны найти все инструкции `switch` и обновить их. Полиморфизм обеспечивает элегантное оружие против темных сил такого повторения.

Циклы

Циклы были фундаментальной частью программирования с самых ранних языков. Но нам кажется, что сегодня они далеко не столь актуальны. Они не нравились нам и во времена первого издания книги, но тогда Java, как и большинство других языков того времени, не предоставляла никакой лучшей альтернативы. Однако в наши дни имеются иные возможности, так что мы можем использовать рефакторинг *Замена цикла конвейером* (с. 278), чтобы удалить эти анахронизмы. Мы считаем, что конвейерные операции, такие как фильтрация и отображение, помогают быстрее увидеть элементы, включенные в обработку, и выяснить, что именно с ними происходит.

Ленивый элемент

Нам нравится использовать программные элементы для улучшения структуры — предоставления возможностей для изменения, повторного использования или просто применения более полезных имен. Но иногда структура не нужна. Это может быть функция с именем, совпадающим с тем, что делает ее тело, или класс, который по сути является одной простой функцией. Иногда это функция, которая должна была вырасти и стать важной и востребованной позже, но которая так и не осуществила свои мечты. Иногда это класс, который раньше был нужен, но оказался уменьшен с помощью рефакторинга. В любом случае, такие элементы программы должны умереть с достоинством. Обычно это означает использование рефакторингов *Встраивание функции* (с. 161) или *Встраивание класса* (с. 232). При наследовании можно воспользоваться рефакторингом *Свертывание иерархии* (с. 423).

Теоретическая общность

Брайан Фут (Brian Foote) предложил название "теоретическая общность" (speculative generality) для запаха, к которому мы очень чувствительны. Он возникает, когда говорят о том, что в будущем, наверное, потребуется возможность делать то или иное, и хотят обеспечить набор механизмов для работы с вещами, которые пока что не нужны. Получающуюся в результате программу труднее понимать и сопровождать. Если бы все эти механизмы использовались, их наличие было бы оправданным, а без этого они только мешают, так что лучше от них избавиться.

Если у вас есть абстрактные классы, не приносящие большой пользы, избавляйтесь от них путем применения рефакторинга *Свертывание иерархии* (с. 423). Излишнее делегирование можно устранить с помощью рефакторингов *Встраивание функции* (с. 161) и *Встраивание класса* (с. 232). Функции с неиспользуемыми параметрами, которые планируется применять в будущем, которое все никак не наступает, должны быть подвергнуты рефакторингу *Изменение объявления функции* (с. 170).

Теоретическая общность может наблюдаться, когда единственными пользователями функции или класса являются тестовые примеры. Найдя такую функцию или класс, удалите тестовый пример и примените рефакторинг *Удаление неработающего кода* (с. 283).

Временное поле

Иногда выясняется, что в некотором классе поле устанавливается только в определенных обстоятельствах. Такой код труден для понимания, поскольку вы ожидаете, что объекту нужны все его поля. Можно сломать голову, пытаясь понять, для чего нужна некоторая переменная, если найти, где она используется, никак не удается.

С помощью рефакторинга *Извлечение класса* (с. 229) создайте приют для бедных осиротевших переменных. Поместите в него весь код, работающий с ними, воспользовавшись рефакторингом *Перенос функции* (с. 244). Возможно, вам удастся удалить условный код с помощью рефакторинга *Введение частного случая* (с. 334), чтобы создать альтернативный класс для случая, когда переменные являются некорректными.

Цепочки сообщений

Цепочки сообщений возникают, когда клиент запрашивает у одного объекта другой, у того клиент в свою очередь запрашивает еще один объект, у которого запрашивается еще один... и т. д. Это может выглядеть как длинный ряд методов типа `getThis` или как последовательность временных переменных. Такие последовательности вызовов означают, что клиент связан со структурой классов, и любые изменения промежуточных связей приводят к необходимости модификации клиента.

Здесь можно применить рефакторинг *Сокрытие делегата* (с. 235), причем его можно использовать в различных местах цепочки. В принципе можно выполнить его для каждого объекта цепочки, но это часто превращает каждый промежуточный объект в посредника. Зачастую лучшей альтернативой оказывается выяснение, для чего используется конечный объект. Посмотрите, нельзя ли с помощью рефакторинга *Извлечение функции* (с. 152) взять использующую его часть кода и с помощью рефакторинга *Перенос функции* (с. 244) сместить вниз по цепочке. Если несколько клиентов одного из объектов цепочки хотят пройти остальную часть пути, добавьте для этого соответствующий метод.

Некоторые программисты рассматривают любую цепочку вызовов методов как нечто ужасное. Авторы же относятся к этому явлению спокойно. По крайней мере в данном случае.

Посредник

Одной из основных характеристик объектов является инкапсуляция — сокрытие внутренних деталей реализации от внешнего мира. Инкапсуляции часто сопутствует делегирование. Например, вы договариваетесь с директором о встрече. Он делегирует это сообщение своему календарю и дает вам ответ. Все хорошо и правильно. Совершенно не важно, использует ли директор ежедневник, электронное устройство или своего секретаря, чтобы вести расписание личных встреч.

Однако такой подход может завести слишком далеко. Например, мы просматриваем интерфейс класса и обнаруживаем, что половина методов делегирует обработку другому классу. В таком случае нужно воспользоваться рефакторингом *Удаление посредника* (с. 238) и общаться с объектом, который действительно знает, что происходит. При наличии нескольких методов, которые не выполняют большой объем работ, их можно поместить в вызывающий метод с помощью рефакторинга *Встраивание функции* (с. 161). При наличии дополнительного поведения с помощью рефакторингов *Замена суперкласса делегатом* (с. 443) или *Замена подкласса делегатом* (с. 424) можно преобразовать посредник в подкласс реального объекта. Это позволит вам расширить поведение, не прибегая к такому делегированию.

Внутренний обмен

Программисты любят высокие крепкие стены между своими модулями и горько жалуются на то, что обмен данными в обход этих стен слишком сильно увеличивает связи. Чтобы заставить программу работать, некоторый обмен необходим, но нам нужно свести его к минимуму и держать под контролем.

Модули, которые перешептываются друг с другом около кофемашины, необходимо разделить с помощью рефакторингов *Перенос функции* (с. 244) и *Перенос поля* (с. 253), чтобы уменьшить необходимость общения. Если модули имеют общие интересы, попробуйте создать третий модуль, чтобы сохранить эту общность в хорошо регулируемом транспортном средстве, или используйте рефакторинг *Сокрытие делегата* (с. 235), чтобы сделать другой модуль посредником.

Наследование часто может привести к сговору. Подклассы всегда будут знать о своих родителях больше, чем их родители хотели бы. Если пришло время дочерним классам уйти из родительского дома, примените рефакторинги *Замена подкласса делегатом* (с. 424) или *Замена суперкласса делегатом* (с. 443).

Большой класс

Когда класс пытается взять на себя слишком многое, это часто проявляется в чрезмерном количестве полей. А отсюда недалеко и до дублирования кода.

Рефакторинг *Извлечение класса* (с. 229) позволяет связать некоторое количество переменных. Выбирайте переменные, которые должны оказаться вместе в компоненте, так, чтобы это имело смысл для каждого из них. Например, `depositAmount` (сумма вклада) и `depositCurrency` (валюта вклада) вполне могут принадлежать одному компоненту. Обычно на мысль о создании компонента наводят одинаковые префиксы или суффиксы у некоторого подмножества членов класса. Если имеет смысл наследование, то зачастую ситуацию могут упростить рефакторинги *Извлечение суперкласса* (с. 418) или *Замена кода типа подклассами* (с. 405).

Иногда класс не использует все свои поля постоянно. В таком случае оказывается возможным неоднократное применение упомянутых рефакторингов.

Как и класс с большим количеством переменных экземпляра, класс со слишком большим количеством кода создает предпосылки для дублирования кода, хаоса и гибели. Простейшее решение (а мы уже не раз говорили, что предпочитаем простые решения) — это устранение избыточности в самом классе. Если у вас есть пять методов по сотне строк и с большим количеством дублируемого кода, возможно, их можно заменить пятью методами по десять и еще десятком двухстрочных методов, выделенных из исходных.

Часто лучшей подсказкой для разделения являются клиенты такого класса. Посмотрите, используют ли клиенты подмножество функций класса. Каждое такое подмножество является возможным отдельным классом. После того как вы определите полезное подмножество, используйте рефакторинги *Извлечение класса* (с. 229), *Извлечение суперкласса* (с. 418) или *Замена кода типа подклассами* (с. 405).

Альтернативные классы с разными интерфейсами

Одним из больших преимуществ использования классов является возможность подстановки, позволяющая в случае необходимости подставлять один класс вместо другого. Но это работает, только если их интерфейсы одинаковы. Используйте рефакторинг *Изменение объявления функции* (с. 170), чтобы сделать функции разных классов совпадающими. Однако зачастую этого оказывается недостаточно; в таком случае продолжайте использовать рефакторинг *Перенос функции* (с. 244) для перемещения поведения в классы, пока их протоколы не совпадут. Если это приводит к дублированию, можете использовать рефакторинг *Извлечение суперкласса* (с. 418).

Классы данных

Такие классы содержат поля, методы для получения и установки значений этих полей и ничего больше. Это молчаливые хранилища данных, которыми другие классы наверняка манипулируют излишне детально. На ранних этапах в этих классах могут быть открытые поля, и тогда необходимо немедленно, пока их никто не обнаружил, применить рефакторинг *Инкапсуляция записи* (с. 208). При наличии полей, которые не должны изменяться, удалите методы установки их значений с помощью рефакторинга *Удаление метода установки значения* (с. 376).

Посмотрите, как методы доступа к полям используются другими классами. Попробуйте использовать рефакторинг *Перенос функции* (с. 244) для перемещения поведения в класс данных. Если функцию не удается переместить целиком, воспользуйтесь рефакторингом *Извлечение функции* (с. 152), чтобы создать функцию, которую можно переместить.

Классы данных часто являются признаком размещения поведения в неправильном месте; это означает, что вы можете добиться большего прогресса, переместив его из клиента в сам класс данных. Но есть исключения, и одно из них — запись, которая используется как результат отдельного вызова функции. Хорошим примером этого является промежуточная структура данных после применения рефакторинга *Разделение этапа* (с. 201). Ключевой характеристикой такой записи результатов является то, что она неизменяема (по крайней мере на практике). Неизменяемые поля не нужно инкапсулировать, а информация, полученная из неизменяемых данных, может быть представлена в виде полей, а не как методы доступа.

Отказ от наследства

Подклассам полагается наследовать методы и данные своих родителей. Но что делать, если это наследство им не нравится или не требуется? И получив все эти дары, они пользуются только малой их частью.

Обычно это означает неправильно продуманную иерархию. Необходимо создать новый ("братский") класс на одном уровне с потомком и использовать рефакторинги *Опускание метода* (с. 403) и *Опускание поля* (с. 404), чтобы вынести в него все неиспользуемые методы. Благодаря этому в родительском классе будет содержаться только та функциональность, которая используется. Часто вы можете встретить совет делать все суперклассы абстрактными.

Мы не будем советовать такие крайности; как минимум этот совет годится не всегда. Мы постоянно обращаемся к созданию подклассов для повторного использования части функций и считаем это совершенно нормальным стилем

программирования. Небольшой запах обычно остается, но не такой уж сильный. Поэтому мы говорим, что, если не принятое наследство вызывает какие-то проблемы, следуйте традиционному совету. Однако не чувствуйте себя обязанным поступать так всегда. В девяти случаях из десяти запах слишком слабый, чтобы требовалось любой ценой от него избавиться.

Запах отвергнутого наследства становится сильнее, если подкласс повторно использует функции суперкласса, но не желает поддерживать его интерфейс. Мы не возражаем против отказа от реализаций, но отказ от интерфейса нас возмущает. В этом случае не возитесь с иерархией; ее лучше разрушить с помощью рефакторингов *Замена подкласса делегатом* (с. 424) и *Замена суперкласса делегатом* (с. 443).

Комментарии

Не волнуйтесь, мы не станем утверждать, что писать комментарии не нужно. В нашей обонятельной аналогии комментарии издают не дурной, а очень даже приятный запах. Мы упомянули комментарии потому, что часто они играют роль дезодоранта. Удивительно часто встречается код с обильными комментариями, которые появились в нем лишь потому, что код плохой.

Комментарии приводят нас к плохому коду, издающему всевозможные дурные запахи, о которых мы писали в этой главе. Первым действием должно быть удаление этих запахов с помощью рефакторинга. После этого комментарии часто оказываются ненужными.

Если для объяснения действий блока кода нужен комментарий, попробуйте применить рефакторинг *Извлечение функции* (с. 152). Если функция уже выделена, но комментарий для объяснения ее действий остается необходимым, воспользуйтесь рефакторингом *Изменение объявления функции* (с. 170). А если требуется изложить некоторые правила, касающиеся необходимого состояния системы, примените рефакторинг *Введение утверждения* (с. 346).

Почувствовав необходимость написать комментарий, попробуйте сначала изменить структуру кода так, чтобы любые комментарии стали ненужными.

Комментарии полезны, когда вы не знаете, что делать. Помимо описания происходящего, комментарии могут отмечать те места, в которых вы не уверены. Комментарии — хорошее место пояснить, *почему* вы поступаете именно так. Эта информация пригодится тем, кто будет работать с вашим кодом в будущем.

Глава 4

Создание тестов

Рефакторинг — ценный инструмент, но он не может существовать в отрыве от тестов. Чтобы правильно выполнить рефакторинг, мне нужен солидный набор тестов, чтобы найти неизбежные ошибки, допущенные мной. Даже при использовании автоматизированных инструментов рефакторинга многие из моих рефакторингов все равно будут нуждаться в проверке с использованием набора тестов.

Я не считаю это недостатком. Даже без рефакторинга написание хороших тестов повышает эффективность моей работы как программиста. Это было неожиданностью для меня и контринтуитивно для большинства программистов, поэтому на этом вопросе стоит остановиться подробнее.

Важность самотестируемого кода

Если вы посмотрите на то, как большинство программистов тратят свое время, то обнаружите, что написание кода на самом деле является довольно малой его частью. Некоторое время тратится на выяснение того, что должно быть сделано, некоторое время тратится на проектирование, но большая часть времени тратится на отладку. Я уверен, что каждый читатель может вспомнить долгие часы отладки, часто до поздней ночи или раннего утра. Каждый программист может рассказать историю ошибки, на поиск которой ушел целый день (а то и больше). Исправление ошибки, как правило, делается довольно быстро, но найти ее — сущий кошмар. А когда вы исправляете ошибку, всегда есть вероятность, что появится другая, которую вы заметите только намного позже, и потратите целую вечность, чтобы найти эту ошибку.

Событием, подтолкнувшим меня на путь создания самотестируемого кода, стал разговор на OOPSLA в 1992 году. Некто (припоминается, это был Дэйв Томас (Dave Thomas)) небрежно заметил: "Классы должны содержать тесты для самих себя". Мне пришло в голову, что это неплохой способ организации тестирования. Я истолковал эти слова так, что в каждом классе должен быть свой метод (например, с именем `test`), с помощью которого класс может протестировать сам себя. Тогда я занимался инкрементной разработкой, а потому попытался по завершении каждого шага добавлять в классы тестирующие методы. Проект был совсем небольшим, и каждый шаг занимал у нас примерно неделю. Выполнять тесты

стало достаточно просто, но хотя их было легко запускать, само тестирование оставалось весьма утомительным, потому что каждый тест выводил на консоль результаты, которые приходилось проверять. Я человек достаточно ленивый, так что готов как следует потрудиться, лишь бы избавиться от лишней работы. Очевидно, что можно не смотреть на экран в поисках некоторой информации, которая должна быть выведена, а заставить заниматься этим компьютер. Все, что требовалось, — это поместить ожидаемые данные в код теста и провести в нем сравнение. После этого можно было просто вызывать тестирующий метод каждого класса, и если все было в порядке, то он просто выводил на экран сообщение OK. Так классы стали самотестируемыми.

Тесты должны быть полностью автоматизированы и проверять свои результаты.

После этого выполнять тесты стало ничуть не труднее, чем компиляцию, так что я начал проводить тестирование при каждой компиляции. Вскоре обнаружилось, что производительность моего труда резко возросла. Я понял, что перестал тратить много времени на отладку! Если я допускал ошибку, перехватываемую предыдущим тестом, это обнаруживалось, как только я запускал этот тест. Поскольку раньше тест работал, я понимал, что ошибка находится там, где я внес изменения после предыдущего тестирования. Тесты я запускал часто, буквально каждые несколько минут, так что всегда знал, где именно находится ошибка — в том коде, который я только что написал. Этот код был еще свеж в памяти, мал по размеру, так что найти ошибку было легко. Часы, которые ранее приходилось тратить на поиск ошибок, превратились в считанные минуты. Такое мощное средство обнаружения ошибок появилось у меня не только благодаря тому, что я писал классы с самотестированием, но и потому, что часто выполнял тестирование.

Придя к таким выводам, я стал тестировать свой код еще более агрессивно. Я стал добавлять тесты сразу же после написания небольших фрагментов функций, не дожидаясь завершения этапа разработки. За день, как правило, добавлялась пара новых функций и тесты для них. На отладку я стал тратить не более нескольких минут в день.

Набор тестов является мощным детектором ошибок, резко сокращающим время их поиска.

Инструменты для написания и организации этих тестов получили большое развитие со времени моих экспериментов. Во время полета из Швейцарии в Атланту на OOPSLA 1997 года Кент Бек в паре с Эрихом Гаммой (Erich Gamma) перенес свою среду модульного тестирования с Smalltalk на Java. Получившийся

каркас под названием JUnit оказал огромное влияние на тестирование программ, вдохновив на создание огромного количества похожих инструментов [41] для множества разных языков.

Убедить других последовать тем же путем, конечно же, было нелегко. Тестам необходим большой объем кода. Самотестирование кажется бессмысленным, пока не убедишься сам, насколько оно ускоряет работу. К тому же многие не просто совершенно не умеют писать тесты, но даже никогда о них и не задумываются. Проводить тестирование вручную — занятие не из веселых, но если его автоматизировать, то написание тестов может даже доставлять удовольствие.

Фактически очень полезно писать тесты еще до начала программирования. Когда требуется добавить новую функциональность, начинайте с создания теста. Это не так глупо, как может показаться. Когда вы пишете тест, то спрашиваете себя, что нужно сделать для добавления этой функциональности. При написании теста вы сосредоточиваетесь на интерфейсе, а не на реализации, что всегда хорошо. Кроме того, это означает наличие точного критерия завершения кодирования — в конечном итоге тест должен успешно проходиться.

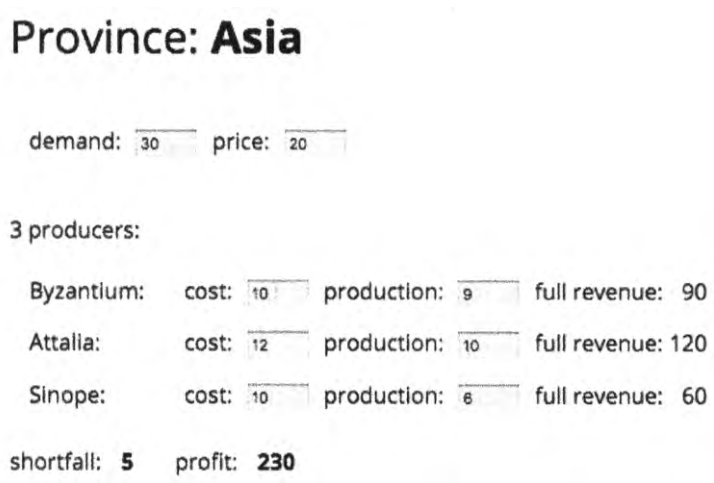
Кент Бек превратил привычку сначала писать тест в методику под названием *Разработка на основе тестов* (Test-Driven Development — TDD) [36]. Такой подход к разработке основывается на коротких циклах написания теста, создания кода, успешно проходящего этот тест, и рефакторинга для обеспечения максимально чистого результата. Этот цикл проверки-кода-рефакторинга должен выполняться много раз в час и может быть очень продуктивным и успокаивающим способом написания кода. Я не собираюсь обсуждать его здесь, но я его использую сам и горячо рекомендую всем.

Но прекратим эту бесполезную дискуссию. Хотя я уверен, что написание самотестируемого кода крайне выгодно для всех, эта книга все же посвящена другой теме — рефакторингу. Рефакторинг требует тестов. Тому, кто собирается заниматься рефакторингом, просто необходимо писать тесты. В этой главе дается беглое представление о том, как это делается при работе на языке программирования JavaScript. Книга посвящена не тестированию, так что я не стану погружаться в детали. Но что касается тестирования, я твердо убедился, что даже малое его количество может принести очень большую пользу.

Как и все остальное в этой книге, подход к тестированию описан с применением большого количества примеров. При разработке кода лично я пишу тесты прямо по ходу работы; но зачастую, когда над рефакторингом работает группа людей, приходится иметь дело с большим объемом кода при отсутствии самотестирования. Поэтому прежде чем применить рефакторинг, нужно сделать код самотестируемым.

Пример кода для тестирования

Вот небольшой код, который нужно просмотреть и проверить. Код поддерживает простое приложение, которое позволяет пользователю изучать производственный план и манипулировать им. (Необработанный) графический интерфейс пользователя выглядит следующим образом.



Производственный план имеет спрос (demand) и цену (price) для каждой географической области. В каждой области есть производители, каждый из которых может производить определенное количество единиц продукции по определенной цене. Интерфейс также показывает, сколько дохода получит каждый производитель, если продаст всю свою продукцию. Внизу экрана показан дефицит производства (спрос минус общий объем производства) и прибыль по этому плану. Пользовательский интерфейс позволяет пользователю манипулировать спросом, ценой, производством и затратами отдельного производителя, чтобы увидеть их влияние на дефицит производства и прибыль. Всякий раз, когда пользователь меняет любое число на дисплее, все остальные значения немедленно обновляются.

Здесь я показал пользовательский интерфейс, чтобы вы могли почувствовать, как используется программное обеспечение, но я планирую сосредоточиться только на бизнес-логике программного обеспечения, т.е. на классах, которые рассчитывают прибыль и дефицит, а не на коде, который генерирует HTML и передает изменения полей к базовой бизнес-логике. Эта глава — просто введение в мир самотестируемого кода, поэтому для меня имеет смысл начать с самого простого случая — кода, который не включает пользовательский интерфейс, сохранение или взаимодействие с внешними службами. Такое разделение является хорошей

идеей в любом случае: как только бизнес-логика усложнится, я отделю ее от механики пользовательского интерфейса, чтобы было легче ее обдумывать и тестировать.

Этот код бизнес-логики включает два класса: класс, представляющий одного производителя, и класс, представляющий целую область. Конструктор области принимает объект JavaScript — тот, который мы могли бы предоставлять в виде документа JSON.

Вот код, который загружает информацию об области из данных JSON:

```
class Province...
constructor(doc) {
  this._name = doc.name;
  this._producers = [];
  this._totalProduction = 0;
  this._demand = doc.demand;
  this._price = doc.price;
  doc.producers.forEach(d=>this.addProducer(new Producer(this,d)));
}

addProducer(arg) {
  this._producers.push(arg);
  this._totalProduction += arg.production;
}
```

Эта функция создает соответствующие данные JSON. Я могу создать образец области для тестирования, создавая объект области из результата следующей функции.

```
Верхний уровень..
function sampleProvinceData() {
  return {
    name: "Asia",
    producers: [
      {name: "Byzantium", cost: 10, production: 9},
      {name: "Attalia",   cost: 12, production: 10},
      {name: "Sinope",    cost: 10, production: 6},
    ],
    demand: 30,
    price: 20
  };
}
```

У класса области есть методы доступа к различным значениям данных.

```
class Province...
get name()               {return this._name;}
get producers()          {return this._producers.slice();}
get totalProduction()    {return this._totalProduction;}
set totalProduction(arg) {this._totalProduction = arg;}
get demand()             {return this._demand;}
```

```
set demand(arg)          {this._demand = parseInt(arg);}
get price()              {return this._price;}
set price(arg)           {this._price = parseInt(arg);}
```

Методы установки вызываются со строками из пользовательского интерфейса, которые содержат числа, поэтому мне нужно выполнить преобразования строк в числа, чтобы использовать их в расчетах.

Класс производителя в основном представляет собой простое хранилище данных:

```
class Producer...
constructor(aProvince, data) {
  this._province = aProvince;
  this._cost = data.cost;
  this._name = data.name;
  this._production = data.production || 0;
}

get name()    {return this._name;}
get cost()    {return this._cost;}
set cost(arg) {this._cost = parseInt(arg);}

get production() {return this._production;}
set production(amountStr) {
  const amount = parseInt(amountStr);
  const newProduction = Number.isNaN(amount) ? 0 : amount;
  this._province.totalProduction+=newProduction - this._production;
  this._production = newProduction;
}
```

Способ, которым `set production` обновляет данные в области, уродлив, и всякий раз, когда я вижу такое, я хочу выполнить рефакторинг, чтобы удалить этот ужас. Но прежде чем я смогу выполнить рефакторинг, следует написать тесты.

Расчет дефицита прост.

```
class Province...
get shortfall() {
  return this._demand - this.totalProduction;
}
```

Расчет прибыли немного более сложен.

```
class Province...
get profit() {
  return this.demandValue - this.demandCost;
}
get demandCost() {
  let remainingDemand = this.demand;
  let result = 0;
  this.producers
    .sort((a,b) => a.cost - b.cost)
```

```
    .forEach(p => {
      const contribution=Math.min(remainingDemand,p.production);
      remainingDemand -= contribution;
      result += contribution * p.cost;
    });
  return result;
}
get demandValue() {
  return this.satisfiedDemand * this.price;
}
get satisfiedDemand() {
  return Math.min(this._demand, this.totalProduction);
}
```

Первый тест

Чтобы протестировать этот код, мне понадобится какой-то каркас тестирования. Их имеется немало, даже только для JavaScript. Я использую довольно распространенный каркас Mocha [43]. Не буду вдаваться в подробное объяснение того, как использовать этот каркас, просто покажу несколько примеров тестов с ним. Вы легко адаптируете используемый вами каркас для создания похожих тестов.

Вот простой тест для расчета дефицита.

```
describe('province', function() {
  it('shortfall', function() {
    const asia = new Province(sampleProvinceData());
    assert.equal(asia.shortfall, 5);
  });
});
```

Каркас Mocha разделяет код теста на блоки, каждый из которых объединяет набор тестов. Каждый тест отображается в блоке `it`. Для данного простого случая тест состоит из двух этапов. На первом этапе настраиваются некоторые "приборы" тестирования — данные и объекты, необходимые для теста; в данном случае в роли такого прибора выступает загруженный объект области. Вторая строка проверяет некоторые характеристики кода; в данном случае, что дефицит соответствует сумме, которую следует ожидать для используемых исходных данных.

Разные разработчики используют описательные строки в блоках `describe` и `it` по-разному. Некоторые пишут пояснение, что именно проверяет тест, другие предпочитают оставлять их пустыми, утверждая, что описательное предложение просто дублирует код так же, как и комментарий. Я предпочитаю небольшое описание, чтобы понимать при сбое, какой тест оказался не пройден.

Когда я запускаю этот тест в консоли NodeJS, вывод выглядит следующим образом:

```
''''''''''''''

1 passing (61ms)
```

Обратите внимание на простоту вывода — просто краткое указание, сколько тестов было выполнено и сколько прошло успешно.

Необходимо убедиться, что тест не даст ложное положительное срабатывание (не покажет, что все в порядке при наличии ошибки).

Когда я пишу тест для существующего кода, как в данном случае, увидеть, что все хорошо, приятно, но я отношусь к этому скептически. В частности, я всегда нервничаю, что тест на самом деле не выполняет код так, как я предполагаю, а потому не обнаружит имеющуюся ошибку. Поэтому я предпочитаю, чтобы каждый тест оказался провален хотя бы один раз. Мой любимый способ сделать это — временно ввести в код ошибку, например:

```
class Province...
get shortfall() {
  return this._demand - this.totalProduction * 2;
}
```

Вот как теперь выглядит вывод на консоль:

```
!

  0 passing (72ms)
  1 failing

  1) province shortfall:
    AssertionError: expected -20 to equal 5
      at Context.<anonymous> (src/tester.js:10:12)
```

Каркас указывает, какой тест не пройден, и дает некоторую информацию о природе сбоя (в данном случае — какое значение ожидалось и какое было получено на самом деле). Поэтому я сразу замечаю, что что-то не так, вижу, какие тесты не пройдены, и это дает мне подсказку о том, что пошло не так (и в данном случае подтверждение того, что проблема именно там, где я ее внес).

Запускайте тесты почаще. Запускайте тесты для кода, над которым вы работаете, хотя бы раз в несколько минут; запускайте все тесты хотя бы раз в день.

В реальной системе могут быть тысячи тестов. Хороший тестовый каркас позволяет легко их запускать и сразу же определять, успешно ли они пройдены. Такая простая обратная связь необходима для самотестирования кода. Работая, я очень часто запускаю тесты для проверки нового кода или наличия ошибок при рефакторинге.

Каркас Mocha может использовать различные библиотеки, которые он называет *библиотеками утверждений* (assertion libraries) для проверки настроек теста. Для JavaScript их насчитывается неимоверное количество. В настоящее время я использую Chai [5], что позволяет мне писать свои проверки в стиле "утверждений":

```
describe('province', function() {
  it('shortfall', function() {
    const asia = new Province(sampleProvinceData());
    assert.equal(asia.shortfall, 5);
  });
});
```

или в стиле "ожиданий":

```
describe('province', function() {
  it('shortfall', function() {
    const asia = new Province(sampleProvinceData());
    expect(asia.shortfall).equal(5);
  });
});
```

Я обычно предпочитаю стиль утверждений, но в настоящее время при работе в JavaScript я в основном использую стиль ожиданий.

Различные среды предоставляют разные способы запуска тестов. Программируя на Java, я использую интегрированную среду разработки, которая предоставляет графический тестер. Индикатор выполнения остается зеленым, когда проходят все тесты, и становится красным в случае неудачи любого из них. Мои коллеги для описания состояния тестов часто используют фразы "зеленая полоса" и "красная полоса". Я мог бы сказать: "Никогда не выполняй рефакторинг на красной полосе", т.е. что не следует приступать к рефакторингу, если в вашем тестовом наборе есть сбойный тест. Или можно сказать "Вернитесь к зеленому", т.е. что вы должны отменить последние изменения и вернуться к последнему состоянию, когда у вас успешно проходил весь набор тестов (обычно это означает вернуться к последней контрольной точке системы управления версиями).

Графические тестеры хороши, но не обязательны. Обычно мои тесты "привязаны" к комбинации клавиш в Emacs, и я наблюдаю текстовую обратную связь в окне компиляции. Ключевым моментом является то, что я могу быстро увидеть, все ли тесты в порядке.

Добавление другого теста

Теперь я продолжу добавлять другие тесты. Стиль, которому я следую, заключается в том, чтобы посмотреть все действия, которые должен выполнять класс, и проверить каждое из них на наличие условий, которые могут привести к сбою работы класса. Это не то же самое, что тестирование каждого открытого метода (подход, который защищают некоторые программисты). Тестирование должно быть ориентировано на риск; я пытаюсь найти ошибки — сейчас или в будущем. Поэтому я не тестирую методы доступа к полям, которые просто читают или записывают поле: они настолько просты, что вряд ли я найду там ошибку.

Это важный момент, так как попытка написать слишком много тестов обычно приводит к тому, что их оказывается недостаточно. Я прочел не одну книгу о тестировании, вызывавшую одно лишь желание любой ценой уклониться от той необъятной работы, которую предлагалось проделать. Всеохватывающее тестирование контрпродуктивно, поскольку заставляет считать, что тестирование всегда связано с чрезмерным трудом. Однако тестирование приносит ощутимую пользу, даже если осуществляется в небольшом объеме. Главное для решения проблемы тестирования — тестировать в основном код, возможность ошибок в котором вызывает наибольшее беспокойство. При этом подходе усилия, затраченные на тестирование, дают максимальную выгоду.

Лучше написать и выполнить неполные тесты, чем не выполнить полные тесты.

Так что я начну с другого основного вывода этого кода — расчета прибыли. И вновь я просто выполняю базовый тест прибыли для исходных данных.

```
describe('province', function() {
  it('shortfall', function() {
    const asia = new Province(sampleProvinceData());
    expect(asia.shortfall).equal(5);
  });

  it('profit', function() {
    const asia = new Province(sampleProvinceData());
    expect(asia.profit).equal(230);
  });
});
```

Здесь показан конечный результат, но способ, которым я его получил, заключался в том, чтобы сначала установить ожидаемое значение в качестве заполнителя, а затем заменить его значением, полученным от программы (230). Я мог бы рассчитать его вручную, но так как код должен работать правильно, сейчас я

просто доверяю ему. После того как этот новый тест заработал для правильного случая, я проверяю сам тест, изменяя расчет прибыли путем добавления ложного умножения на 2. Я убеждаюсь, что тест (как и следует) проваливается, а затем убираю введенную мною ошибку. Этот шаблон — запись с использованием заполнителя для ожидаемого значения, замена заполнителя фактическим значением, возвращаемым кодом, введение ошибки, восстановление корректного кода — является наиболее распространенной моделью, которую я использую при добавлении тестов в существующий код.

Между этими тестами есть определенное дублирование — они оба работают с одной и той же первой строкой. Так же, как я с подозрением отношусь к дублированию в обычном коде, я с подозрением отношусь к нему и в тестовом коде, и поэтому постараюсь удалить его. Один из вариантов заключается в поднятии константы во внешнюю область видимости.

```
describe('province', function() {
  const asia = new Province(sampleProvinceData()); // Не делай так
  it('shortfall', function() {
    expect(asia.shortfall).equal(5);
  });

  it('profit', function() {
    expect(asia.profit).equal(230);
  });
});
```

Но, как указывает комментарий, я никогда так не делаю. В данный момент это сработает, но на самом деле это рассадник самых неприятных ошибок в тестировании — совместно используемый объект, который заставляет тесты взаимодействовать. Ключевое слово `const` в JavaScript означает только константность ссылки на `asia`, а не константность содержимого этого объекта. Если будущий тест изменит этот общий объект, то в конечном итоге могут появляться периодические сбои тестов из-за их взаимодействия через общие объекты, что может приводить к разным результатам в зависимости от порядка выполнения тестов. Этот недетерминизм в тестах может привести к долгой и трудной отладке в лучшем случае и полному недоверию к тестам — в худшем. Вместо этого я предпочитаю поступать следующим образом.

```
describe('province', function() {
  let asia;
  beforeEach(function() {
    asia = new Province(sampleProvinceData());
  });

  it('shortfall', function() {
    expect(asia.shortfall).equal(5);
  });
```

```
  it('profit', function() {
    expect(asia.profit).equal(230);
  });
});
```

Конструкция `beforeEach` запускается перед каждым тестом, сбрасывая объект `asia` и устанавливая каждый раз новое значение переменной. Таким образом, перед каждым тестом создается новый объект, что делает тесты изолированными один от другого и предотвращает недетерминированность, вызывающую столько проблем.

Когда я даю этот совет, некоторые программисты беспокоятся, не будет ли создание нового объекта каждый раз тормозить тесты. В большинстве случаев это будет незаметно. Если же это создает проблему, я бы рассмотрел возможность совместного использования объекта, но с большими предосторожностями, чтобы ни один тест никогда его не изменял. Прибегнуть к совместно используемому объекту можно при гарантии, что он действительно неизменный. Но мой рефлекс состоит в том, чтобы использовать новый объект, потому что в прошлом у меня были слишком большие неприятности, связанные с ошибкой из-за совместно используемого объекта.

Учитывая, что я запускаю код настройки в `beforeEach` для каждого теста, почему бы не оставить его внутри отдельных блоков `it`? Но мне нравится, когда все мои тесты работают с одним и тем же объектом. Наличие блока `beforeEach` сигнализирует читателю кода, что я использую стандартный объект для тестирования. Затем вы можете просмотреть все тесты в рамках этого блока `describe`, зная, что они принимают одни и те же базовые данные.

Изменение прибора тестирования

До сих пор написанные мною тесты показывали, как я проверяю свойства прибора тестирования — объекта `asia` — после его загрузки. Но при использовании этот объект будет регулярно обновляться пользователями по мере изменения значений.

Большинство обновлений являются простыми методами установки значений, и я обычно не пытаюсь их тестировать в силу крайне малой вероятности, что они станут источником ошибки. Но у метода установки `production` достаточно сложное поведение, так что, я думаю, его стоит проверить.

```
describe('province'...
  it('change production', function() {
    asia.producers[0].production = 20;
    expect(asia.shortfall).equal(-6);
    expect(asia.profit).equal(292);
  });
```

Это распространенный шаблон. Я беру исходный стандартный прибор, *настроенный* блоком `beforeEach`, выполняю *тестирование*, затем *проверяю*, что сделано именно то, что должно быть сделано. Если вы прочтете о тестировании побольше, то увидите эту последовательность, возможно, описанную другими словами. Иногда вы увидите все шаги в самом тесте, в других случаях некоторые ранние фазы могут быть перенесены в стандартные процедуры настройки, такие как `beforeEach`.

(Есть еще один неявный четвертый этап, который обычно не упоминается: *удаление*, которое удаляет прибор между тестами, чтобы разные тесты не взаимодействовали друг с другом. Выполняя все настройки в `beforeEach`, я позволяю тестовому каркасу неявно удалять мой тестирующий прибор между тестами, так что я применяю эту фазу неявно. Большинство авторов тестов также задействуют ее неявно, но иногда может оказаться важным иметь явную операцию удаления, особенно если наш прибор совместно используется разными тестами в силу сложности/длительности его создания.)

В этом тесте я проверяю две разные характеристики в одной конструкции `it`. В общем случае целесообразно иметь только одну инструкцию проверки в каждой конструкции `it`. Это связано с тем, что сбойный тест может скрывать полезную информацию, когда вы выясняете, почему именно тест не пройден. В этом случае я чувствую, что проверки достаточно тесно взаимосвязаны, и я проверяю их в одном тесте. Если я захочу разделить их на отдельные конструкции `it`, то смогу сделать это позже.

Проверка границ

До сих пор мои тесты были сосредоточены на регулярном использовании, часто именуемом "счастливым путем", когда все идет хорошо и все работает, как и ожидалось. Но стоит также проводить тесты и на границах этих условий — чтобы увидеть, что происходит, когда что-то может пойти не так.

Всякий раз, когда у меня есть коллекция чего-либо (в данном примере — производителей), я предпочитаю проверить, что происходит, когда такая коллекция пуста.

```
describe('no producers', function() {
  let noProducers;
  beforeEach(function() {
    const data = {
      name: "No producers",
      producers: [],
      demand: 30,
      price: 20
    };
```

```
    noProducers = new Province(data);
  });
  it('shortfall', function() {
    expect(noProducers.shortfall).equal(30);
  });
  it('profit', function() {
    expect(noProducers.profit).equal(0);
  });
```

В случае обычных чисел неплохо бы проверить как нулевые значения:

```
describe('province'...
  it('zero demand', function() {
    asia.demand = 0;
    expect(asia.shortfall).equal(-25);
    expect(asia.profit).equal(0);
  });
```

так и отрицательные:

```
describe('province'...
  it('negative demand', function() {
    asia.demand = -1;
    expect(asia.shortfall).equal(-26);
    expect(asia.profit).equal(-10);
  });
```

В этот момент я могу задаться вопросом, действительно ли отрицательный спрос, приводящий к отрицательной прибыли, имеет какой-то смысл для данной предметной области? Не должен ли минимальный спрос быть нулевым? В этом случае, возможно, метод установки значения должен реагировать на отрицательный аргумент иначе — выдавая ошибку или устанавливая значение равным нулю. Это хорошие вопросы, и написание подобных тестов помогает мне подумать о том, как код должен реагировать на граничные случаи.

Подумайте о граничных условиях, которые могут быть неправильно обработаны, и сосредоточьте на них свои усилия.

Методы установки значений берут из полей в графическом интерфейсе пользователя строки, которые, хотя и ограничены только числами, все еще могут быть пустыми, а потому у меня должны быть тесты, которые гарантируют, что код реагирует на пустые строки так, как мне нужно.

```
describe('province'...
  it('empty string demand', function() {
    asia.demand = "";
    expect(asia.shortfall).NaN;
    expect(asia.profit).NaN;
  });
```

Обратите внимание, как я играю роль врага моего кода. Я активно размышляю о том, как мне его поломать. Я нахожу это не только полезным, но и веселым (это соответствует подлым наклонностям моей души).

Это интересно:

```
describe('string for producers', function() {
  it('', function() {
    const data = {
      name: "String producers",
      producers: "",
      demand: 30,
      price: 20
    };
    const prov = new Province(data);
    expect(prov.shortfall).equal(0);
  });
```

Это не приводит к простому сообщению об отказе, что дефицит не равен 0. Вот вывод на консоль:

```
''''''''!

  9 passing (74ms)
  1 failing

  1) string for producers :
    TypeError: doc.producers.forEach is not a function
      at new Province (src/main.js:22:19)
      at Context.<anonymous> (src/tester.js:86:18)
```

Каркас Mocha рассматривает это как сбой, но многие тестовые среды различают данную ситуацию, которую они называют ошибкой, и регулярный сбой. Сбой (failure) означает шаг проверки, когда фактическое значение выходит за границы, ожидаемые инструкцией проверки. Но **ошибка** (error) — дело другое, это исключение, сгенерированное на более ранней стадии (в данном случае на этапе настройки). Она выглядит как исключение, которого авторы кода не ожидали, поэтому мы получаем хорошо знакомую программистам на JavaScript ошибку ("... не является функцией").

Как код должен реагировать на такой случай? Один из подходов состоит в том, чтобы добавить некоторую обработку, которая дала бы лучшую реакцию на ошибку — либо выводя более информативное сообщение об ошибке, либо просто делая `producers` пустым массивом (возможно, с записью соответствующего сообщения в журнальный файл). Но могут быть и веские причины оставить все как есть. Возможно, входной объект создается доверенным источником, таким как другая часть той же кодовой базы. Включение большого количества проверок между модулями в одной и той же кодовой базе может привести к дублированию проверок, что вызовет больше проблем, чем пользы, особенно если они

дублируют проверки, выполняемые в другом месте. Но если этот входной объект поступает из внешнего источника, такого как запрос в кодировке JSON, проверки корректности необходимы, и они должны быть выполнены. В любом случае, написание тестов приводит к такого рода вопросам.

Если бы я писал подобные тесты перед рефакторингом, вероятно, я бы отказался от этого теста. Рефакторинг должен сохранять наблюдаемое поведение; данная же ошибка выходит за границы наблюдаемого поведения, поэтому мне не нужно беспокоиться, если выполняемый мною рефакторинг изменит реакцию кода на данное условие.

Если эта ошибка может привести к некорректным данным и сбою программы, который будет трудно отладить, можно воспользоваться рефакторингом *Введение утверждения* (с. 346). Я не добавляю тесты для выявления таких сбоев утверждений, поскольку они сами по себе являются разновидностью тестирования.

Опасения по поводу того, что тестирование не выявит все ошибки, не должно мешать писать тесты, которые выявят большинство ошибок.

Когда нужно остановиться? Наверняка вы не раз слышали, что тестирование не доказывает отсутствие ошибок в программе. Это верно, но это не мешает тестированию повысить скорость работы программиста. Было предложено немало правил, призванных гарантировать тестируемость всех мыслимых комбинаций данных. Ознакомиться с ними полезно, но не стоит принимать их слишком всерьез. Они могут уменьшить выгоду, приносимую тестированием, а кроме того, имеется опасность, что попытка написать слишком много тестов оставит в душе программиста такой след, что в итоге он вообще перестанет писать тесты. Необходимо сконцентрироваться на наиболее подозрительных местах. Изучите код и определите, где он становится сложным. Изучите функцию и установите области, где могут возникнуть ошибки. Тесты не выявят все ошибки, но в процессе рефакторинга помогут лучше понять программу и благодаря этому найти больше ошибок. Я уже говорил, что всегда начинаю рефакторинг с создания комплекта тестов; по мере продвижения вперед я обязательно пополняю этот комплект.

И многое другое...

Это все, что я собирался сказать в данной главе — в конце концов, эта книга о рефакторинге, а не о тестировании. Но тестирование — важная тема, как потому, что это необходимая основа для рефакторинга, так и потому, что это ценный инструмент сам по себе. Хотя я был рад видеть, что со времени написания первого издания этой книги рефакторинг стал повседневной практикой

программирования, я был еще более счастлив увидеть изменение отношения к тестированию. Ранее считавшееся обязанностью отдельной (зачастую подчиненной) группы, ныне тестирование становится все более первостепенной задачей любого достойного разработчика программного обеспечения. Архитектуры часто справедливо оцениваются по возможности их тестирования.

Типы тестов, которые я показал в этой главе, — это модульные тесты, разработанные для работы с небольшими фрагментами кода и быстрого выполнения. Они являются основой самотестируемого кода; большинство тестов в такой системе являются юнит-тестами. Существуют и другие разновидности тестов, сосредоточенные на интеграции между компонентами, использующие одновременно несколько уровней программного обеспечения, находящие проблемы с производительностью и т.д.

Как и большинство аспектов программирования, тестирование — действие итеративное. Если только вы не очень опытны или не очень удачливы, вы не сможете получить правильные тесты с первого раза. Я обнаружил, что постоянно работаю над набором тестов так же, как и над основным кодом. Естественно, это означает добавление новых тестов наряду с добавлением новых функциональных возможностей, но сюда входит и пересмотр существующих тестов. Достаточно ли они ясны? Не стоит ли их реорганизовать, чтобы было легче понять, что они делают? Есть ли у меня необходимые правильные тесты? Очень хорошая привычка — реагировать на ошибку путем написания теста, который четко ее выявляет. Только после создания такого теста я могу исправить ошибку. Имея тест, я знаю, остается ошибка в коде или она уже исправлена. Я также пытаюсь понять: не может ли эта ошибка и ее тест дать подсказку о других возможных проблемах?

Получив сообщение об ошибке, начните с написания модульного теста, который ее выявляет.

Распространенным вопросом является следующий — "Сколько тестов достаточно?" Некоторые разработчики рекомендуют использовать в качестве меры покрытие тестами [35], но анализ такого покрытия хорош только для определения непроверенных областей кода, но не для оценки качества набора тестов.

Лучшая мера для достаточно хорошего набора тестов — субъективная: насколько вы уверены, что если кто-то внесет дефект в код, то тесты это выявят? Это не тот показатель, который можно объективно проанализировать, и он не учитывает ложную уверенность, но цель самотестируемого кода — в получении такой уверенности. Если я смогу реорганизовать свой код и быть уверенным, что я не внес никаких ошибок, потому что мои тесты остаются "зелеными" — я могу быть доволен своим набором тестов.

Можно написать и слишком много тестов. Одним из признаков этого является то, что я трачу больше времени на изменение тестов, чем на тестируемый код, и чувствую, что возня с тестами не ускоряет, а замедляет мою работу. Но хотя избыточное тестирование и случается, оно является исчезающе редким по сравнению с тестированием недостаточным.

Глава 5

На пути к каталогу рефакторингов

Остальная часть этой книги представляет собой каталог рефакторингов. Этот каталог начинался с моих личных заметок, которые я сделал, чтобы напомнить себе, как выполнять тот или иной рефакторинг безопасным и эффективным способом. С тех пор я доработал каталог, и он стал куда бо́льшим благодаря преднамеренному изучению ряда рефакторингов. Я сам прибегаю к этому каталогу, когда выполняю рефакторинг, к которому давно не обращался.

Формат описания рефакторинга

При описании рефакторингов в этой и других главах соблюдается стандартный формат. Описание каждого метода состоит из пяти частей, как показано ниже.

- Первым идет **название**. Оно играет роль при создания списка методов рефакторинга и используется повсюду в книге.
- За названием следуют **краткая сводка** ситуаций, в которых применим данный метод, и краткие сведения о том, что именно он делает. Это позволяет быстрее находить необходимый метод.
- **Мотивация** описывает, почему следует пользоваться этим методом рефакторинга и когда не следует его применять.
- **Техника** подробно, шаг за шагом, описывает, как применять данный рефакторинг.
- **Примеры** содержат иллюстрацию (очень простого) применения метода, демонстрирующую его действие.

Краткая сводка демонстрирует пример кода, трансформируемого с помощью рефакторинга. Она не предназначена для объяснения того, что такое данный рефакторинг, не говоря уже о том, как его выполнять, но она должна помочь вам вспомнить этот рефакторинг, если вы уже сталкивались с ним раньше. Если нет — то вам, вероятно, придется проработать пример, чтобы получить лучшее

представление о рефакторинге. Я также включаю небольшую графику, которая также предназначена не для пояснений, а для напоминания.

Описание техники основано на моих заметках о том, как выполнять данный рефакторинг. Они позволяют мне вспомнить рефакторинг, которым я некоторое время не пользовался. Поэтому данный раздел, как правило, краток и не поясняет, почему следует выполнять те или иные шаги. Более подробные пояснения можно найти в примере. Таким образом, описание техники представляет собой просто краткие заметки, к которым можно обратиться, когда данный рефакторинг вам знаком, но следует освежить память (так, по крайней мере, использую их я). При проведении рефакторинга впервые вам, вероятно, потребуется полностью прочесть пример.

Я описывал технику таким образом, чтобы каждый шаг рефакторинга был как можно мельче. Особое внимание при проведении рефакторинга уделяется вопросам осторожности, поэтому рефакторинг следует выполнять маленькими шажками, проводя после каждого из них тестирование. На практике я обычно выполняю более крупные шаги, чем описано здесь; столкнувшись с ошибкой, я отступаю назад и двигаюсь более мелкими шажками. При описании шагов можно найти ряд ссылок на особые случаи. Таким образом, описание шагов выступает также в качестве контрольного списка (зачастую я и сам забываю об этих вещах).

Хотя я (за редким исключением) привожу только один вариант техники — она не является единственным способом проведения рефакторинга. В книге я описываю конкретную технику, выбранную среди возможных вариантов, потому что большую часть времени она работает довольно хорошо. Скорее всего, по мере накопления опыта в рефакторинге вы будете менять технику — и это нормально. Просто помните, что главное — делать маленькие шаги, и, чем сложнее ситуация, тем меньшими они должны быть.

Примеры просты до смешного. Их задача — помочь разъяснению рефакторинга, но при этом как можно меньше отвлекать читателя на не относящиеся к делу подробности (надеюсь, что по этой причине их упрощенность простительна; они, несомненно, не могут служить примерами добротного проектирования бизнес-объектов). Я уверен, что, разобравшись, вы сможете применить рефакторинг к своим более сложным исходным текстам. Для некоторых совсем уж простых методов рефакторинга примеры отсутствуют, поскольку вряд ли от них могло бы быть много пользы.

Не забывайте, что эти примеры включены для иллюстрации только одного обсуждаемого метода рефакторинга. В большинстве случаев в получаемом в результате рефакторинга коде сохраняются проблемы, для решения которых требуется применение других методов рефакторинга. В ряде случаев, когда методы рефакторинга часто комбинируются один с другим, я переношу примеры из

одного рефакторинга в другой. При этом чаще всего я оставляю код таким, каким он получается после первого рефакторинга. Это сделано для того, чтобы каждый рефакторинг был автономным (ведь каталог рефакторингов в первую очередь должен служить справочником).

Выбор рефакторинга

Это ни в коем случае не полный каталог рефакторингов. Я лишь надеюсь, что это хотя бы коллекция наиболее полезных рефакторингов. Под "наиболее полезными" я подразумеваю те, которые используются чаще других и которые стоит именовать и описать. Я нахожу что-то стоящим описания по ряду причин: у некоторых интересная техника, которая помогает приобретению общих навыков рефакторинга, другие сильно влияют на улучшение проекта кода.

Некоторые рефакторинги отсутствуют, потому что они настолько малы и прямолинейны, что мне кажется, о них не стоит и писать. Примером может служить отсутствующий в первом издании рефакторинг *Перемещение инструкций* (с. 269), который я часто использую, но который не счел в тот момент достойным включения в каталог. Такие рефакторинги могут быть добавлены в книгу со временем, в зависимости от того, сколько энергии я затрачу на новые рефакторинги в будущем.

Другая категория — это рефакторинги, которые логически существуют, но я либо мало ими пользуюсь, либо они демонстрируют сильное сходство с другими рефакторингами. Каждый рефакторинг в этой книге имеет логически обратный ему рефакторинг, но я не записал их все, потому что не считаю каждую инверсию интересной. Так, рефакторинг *Инкапсуляция переменной* (с. 178) является распространенным и мощным рефакторингом, но его инверсия — то, что я почти никогда не делаю (да и в любом случае ее легко выполнить), поэтому в каталоге соответствующего рефакторинга нет.

Глава 6

Первое множество рефакторингов

Я начинаю каталог с набора рефакторингов, которые считаю наиболее полезными для изучения в первую очередь.

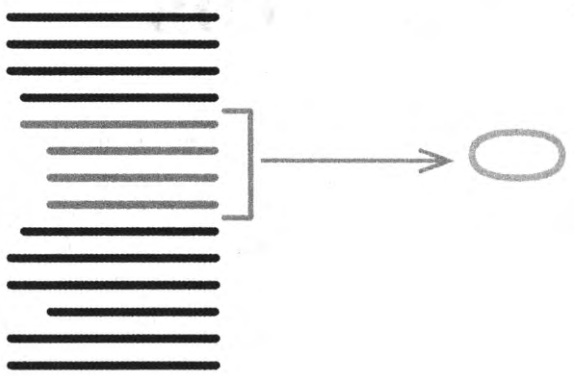
Вероятно, наиболее распространенные рефакторинги, которые я выполняю, — это *Извлечение функции* (с. 152) и *Извлечение переменной* (с. 178). Поскольку любой рефакторинг связан с изменениями, неудивительно, что я также часто использую инверсии этих рефакторингов, а именно — *Встраивание функции* (с. 161) и *Встраивание переменной* (с. 169).

Извлечение тесно связано с присваиванием имен, и (особенно во время учебы) мне часто приходится менять имена. Имена функций меняет рефакторинг *Изменение объявления функции* (с. 170). Я также использую этот рефакторинг для добавления или удаления аргументов функции. Для переменных я применяю рефакторинг *Переименование переменной* (с. 183), который опирается на рефакторинг *Инкапсуляция переменной* (с. 178). При изменении аргументов функции я часто нахожу полезным объединить общий набор аргументов в единый объект с помощью рефакторинга *Введение объекта параметра* (с. 186).

Формирование и именование функций являются существенно низкоуровневыми рефакторингами, но затем необходимо сгруппировать функции в высокоуровневые модули. Я использую рефакторинг *Объединение функций в класс* (с. 190), чтобы сгруппировать функции вместе с данными, с которыми они работают, в класс. Другой путь состоит в применении рефакторинга *Объединение функций в преобразование* (с. 195), что особенно удобно для данных, предназначенных только для чтения. На более высоком уровне эти модули можно преобразовать в отдельные этапы обработки, используя рефакторинг *Разделение этапа* (с. 201).

Извлечение функции (Extract Function)

Бывший рефакторинг *Извлечение метода*
Обратный к рефакторингу *Встраивание функции* (с. 161)



```
function printOwing(invoice) {
  printBanner();
  let outstanding = calculateOutstanding();

  // Вывод детальной информации
  console.log(`name: ${invoice.customer}`);
  console.log(`amount: ${outstanding}`);
}
```

```
function printOwing(invoice) {
  printBanner();
  let outstanding = calculateOutstanding();
  printDetails(outstanding);

  function printDetails(outstanding) {
    console.log(`name: ${invoice.customer}`);
    console.log(`amount: ${outstanding}`);
  }
}
```

Мотивация

Это один из наиболее часто выполняемых мною рефакторингов. (Я использую здесь термин “функция”, но все то же самое справедливо и для метода в объектно-ориентированном языке программирования, а также для любой разновидности процедуры или подпрограммы.) Я нахожу фрагмент кода, разбираюсь, как

он работает, и преобразую этот фрагмент кода в отдельную функцию с именем, указывающим ее предназначение.

За свою карьеру я слышал много споров о том, когда следует выделять код в отдельную функцию. Некоторые из этих рекомендаций основаны на длине кода: функции должны быть небольшими, полностью помещающимися на экране. Другие основаны на повторном использовании: любой код, применяемый более одного раза, должен быть помещен в отдельную функцию, но код, задействованный однократно, должен оставаться нетронутым. Однако для меня наибольшее значение имеет иной аргумент — разделение между намерением и реализацией. Если вам приходится тратить время и силы на просмотр фрагмента кода и выяснение, *что* он делает, — вы должны извлечь его в функцию и назвать функцию этим "*что*". Теперь, когда вы будете читать его снова, назначение функции будет бросаться в глаза, и вам не придется тратить большую часть времени на понимание, как функция выполняет свое предназначение (т.е. на тело функции).

Как только я принял этот принцип, у меня появилась привычка писать очень маленькие функции — обычно всего в несколько строк. Для меня любая функция с более чем десятком строк кода начинает дурно пахнуть, и у меня в коде не редкость функции, представляющие собой единственную строку кода. Тот факт, что размер не важен, был доведен до моего сведения примером, который Кент Бек показал мне в исходной системе Smalltalk. В те времена Smalltalk работал на черно-белых мониторах. Если вы хотели выделить какой-либо текст или графику, его требовалось инвертировать. Графический класс Smalltalk имел для этого метод, называемый `highlight`, реализация которого была просто вызовом метода `reverse`. Имя метода было длиннее, чем его реализация, но это не имело значения, потому что между намерением кода и его реализацией было большое расстояние.

Некоторые программисты волнуются из-за коротких функций, потому что беспокоятся о производительности при их вызове. Когда я был молодым, это иногда действительно играло роль, но сейчас такое встречается крайне редко. Оптимизирующие компиляторы часто куда лучше работают с короткими функциями, которые легче кешировать. Как всегда, просто следуйте общим рекомендациям по оптимизации производительности.

Небольшие функции, подобные упомянутой, имеют смысл только в случае, если их имена хороши, поэтому вам следует уделить особое внимание именам. Это требует практики; но как только вы получите достаточный опыт — такой подход может сделать код самодокументированным.

Нередко я вижу в крупных функциях фрагменты кода, которые начинаются с комментария, рассказывающего, что они делают. Комментарий часто является хорошей подсказкой для имени функции, когда я извлекаю этот фрагмент в отдельную функцию.

Техника

- Создайте новую функцию и дайте ей имя, соответствующее ее предназначению (тому, *что* она делает, а не *как*).

  Когда предполагаемый для выделения код очень прост (например, выводит отдельное сообщение или вызывает одну функцию), его стоит извлекать и выделять в отдельную функцию, только если имя новой функции лучше раскрывает назначение кода. Если вы не можете придумать содержательное имя, не извлекайте такой код. Однако мне не нужно сразу придумывать лучшее имя; иногда хорошее имя появляется только тогда, когда я работаю с извлечением функции. Можно извлечь функцию, попытаться поработать с ней, понять, что это было сделано зря, и восстановить status quo. Пока я узна*ю* что-то новое — мое время не тратится впустую. Если язык поддерживает вложенные функции, вложите извлеченную функцию в исходную функцию. Это уменьшит количество переменных, выходящих за пределы области видимости, с которыми после придется иметь дело. Я всегда могу использовать рефакторинг *Перенос функции* (с. 244) позже.
- Скопируйте код, подлежащий извлечению, из исходной функции в создаваемую.
- Найдите в извлеченном коде все обращения к переменным, имеющим локальную область видимости в исходной функции и которые не будут находиться в области видимости извлеченной функции. Передавайте их в качестве параметров.

  Если я извлекаю код в функцию, вложенную в исходную, то я не сталкиваюсь с этими проблемами.

  Зачастую это локальные переменные и параметры функции. Наиболее общий подход заключается в передаче всех таких параметров в качестве аргументов. Обычно нет никаких проблем с переменными, которые используются, но не присвоены.

  Если переменная используется только внутри извлеченного кода, но объявлена снаружи, переместите ее объявление в извлеченный код.

  Любые присвоенные переменные нуждаются в большей аккуратности, если они передаются по значению. Если имеется только одна такая переменная, я пытаюсь рассматривать извлеченный код как запрос и присваивать его результат соответствующей переменной.

  Иногда я обнаруживаю, что извлеченный код присваивает значения слишком большому количеству локальных переменных. На этом этапе лучше отказаться от извлечения функции. В этой ситуации я

рассматриваю другие рефакторинги, такие как *Расщепление переменной* (с. 285) или *Замена временной переменной запросом* (с. 225), чтобы упростить использование переменной и вернуться к извлечению позже.

- Выполните компиляцию после обработки всех переменных.

  После обработки всех переменных может быть полезно выполнить компиляцию, если языковая среда выполняет проверки времени компиляции. Часто это помогает найти любые переменные, которые не были должным образом обработаны.

- Замените в исходной функции извлеченный код вызовом целевой функции.
- Выполните тестирование.
- Найдите другой код, идентичный только что извлеченному или похожий на него, и подумайте об использовании рефакторинга *Замена встроенного кода вызовом функции* (с. 268) для вызова новой функции.

  Некоторые инструменты рефакторинга непосредственно поддерживают такое преобразование. В противном случае имеет смысл выполнить несколько быстрых поисков, чтобы увидеть, нет ли дублируемого кода в другом месте.

Пример: переменных вне области видимости нет

В простейшем случае данный рефакторинг тривиален.

```
function printOwing(invoice) {
  let outstanding = 0;

  console.log("************************");
  console.log("**** Customer Owes ****");
  console.log("************************");

  // Выполнение расчетов
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  // Запись срока оплаты
  const today = Clock.today;
  invoice.dueDate = new Date(today.getFullYear(),
                             today.getMonth(),
                             today.getDate() + 30);
  // Вывод детальной информации
  console.log(`name: ${invoice.customer}`);
  console.log(`amount: ${outstanding}`);
  console.log(`due: ${invoice.dueDate.toLocaleDateString()}`);
}
```

Вас может заинтересовать, что такое `Clock.today`. Это Clock Wrapper [21] — объект, который служит "оберткой" для обращений к системным часам. В моем коде я избегаю таких прямых вызовов, как `Date.now()`, потому что это приводит к недетерминированным тестам и затрудняет воспроизведение состояний ошибок при диагностике сбоев.

Легко извлечь код, который печатает заголовок. Я просто вырезаю код, выполняю его вставку и вставку вызова.

```
function printOwing(invoice) {
  let outstanding = 0;

  printBanner();

  // Выполнение расчетов
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  // Запись срока оплаты
  const today = Clock.today;
  invoice.dueDate = new Date(today.getFullYear(),
                             today.getMonth(),
                             today.getDate() + 30);
  // Вывод детальной информации
  console.log(`name: ${invoice.customer}`);
  console.log(`amount: ${outstanding}`);
  console.log(`due: ${invoice.dueDate.toLocaleDateString()}`);
}

function printBanner() {
  console.log("************************");
  console.log("**** Customer Owes ****");
  console.log("************************");
}
```

Точно так же я могу извлечь вывод детальной информации:

```
function printOwing(invoice) {
  let outstanding = 0;

  printBanner();

  // Выполнение расчетов
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  // Запись срока оплаты
  const today = Clock.today;
  invoice.dueDate = new Date(today.getFullYear(),
                             today.getMonth(),
                             today.getDate() + 30);
```

```
  printDetails();

  function printDetails() {
    console.log(`name: ${invoice.customer}`);
    console.log(`amount: ${outstanding}`);
    console.log(`due: ${invoice.dueDate.toLocaleDateString()}`);
  }
```

Это делает извлечение функции выглядящей как тривиально простой рефакторинг. Но часто ситуация оказывается куда более сложной.

В показанном выше случае я определил `printDetails` так, чтобы эта функция была вложена в `printOwing`. Таким образом она может получить доступ ко всем переменным, определенным в `printOwing`. Но этот вариант не годится, если я программирую на языке, который не допускает вложенные функции. В этом случае следует извлечь функцию на верхний уровень, т.е. следует обращать внимание на любые переменные, которые существуют в области видимости исходной функции. Это аргументы исходной функции и временные переменные, определенные в ней.

Пример: использование локальных переменных

Самый простой случай с локальными переменными — когда они используются, но не переприсваиваются. В этом случае их можно передать в качестве параметров. Таким образом, если у меня есть следующая функция:

```
function printOwing(invoice) {
  let outstanding = 0;

  printBanner();

  // Выполнение вычислений
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  // Запись срока оплаты
  const today = Clock.today;
  invoice.dueDate = new Date(today.getFullYear(),
                             today.getMonth(),
                             today.getDate() + 30);

  // Вывод детальной информации
  console.log(`name: ${invoice.customer}`);
  console.log(`amount: ${outstanding}`);
  console.log(`due: ${invoice.dueDate.toLocaleDateString()}`);
}
```

то я могу извлечь функцию вывода детальной информации, передавая ей два параметра:

```
function printOwing(invoice) {
  let outstanding = 0;

  printBanner();

  // Выполнение вычислений
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  // Запись срока оплаты
  const today = Clock.today;
  invoice.dueDate = new Date(today.getFullYear(),
                             today.getMonth(),
                             today.getDate() + 30);

  printDetails(invoice, outstanding);
}

function printDetails(invoice, outstanding) {
  console.log(`name: ${invoice.customer}`);
  console.log(`amount: ${outstanding}`);
  console.log(`due: ${invoice.dueDate.toLocaleDateString()}`);
}
```

То же самое верно, если локальная переменная является структурой (такой как массив, запись или объект), и я изменяю эту структуру. Таким образом, я могу аналогичным образом извлечь настройки срока оплаты.

```
function printOwing(invoice) {
  let outstanding = 0;

  printBanner();

  // Выполнение вычислений
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  recordDueDate(invoice);
  printDetails(invoice, outstanding);
}

function recordDueDate(invoice) {
  const today = Clock.today;
  invoice.dueDate = new Date(today.getFullYear(),
                             today.getMonth(),
                             today.getDate() + 30);
}
```

Пример: присваивание локальной переменной

Наибольшие трудности возникают, когда нужно присвоить локальным переменным новые значения. Сейчас мы говорим только о временных переменных. Если вы увидите присваивание параметру, то должны сразу же применить рефакторинг *Расщепление переменной* (с. 285).

Есть два варианта присваивания временным переменным. В простейшем случае временная переменная используется лишь внутри извлекаемого кода. Тогда временную переменную можно переместить в этот извлекаемый код. Иногда, в частности, когда переменные инициализируются на некотором расстоянии до своего применения, удобно использовать рефакторинг *Перемещение инструкций* (с. 269), чтобы собрать всю работу с переменной в одно место.

Более сложен случай, когда переменная используется вне извлеченной функции. В этом случае мне нужно вернуть новое значение. Я могу проиллюстрировать это с помощью следующей знакомой функции.

```
function printOwing(invoice) {
  let outstanding = 0;

  printBanner();

  // Выполнение вычислений
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  recordDueDate(invoice);
  printDetails(invoice, outstanding);
}
```

Все предыдущие рефакторинги я выполнял за один шаг, поскольку они были очень простыми, но на этот раз я буду работать пошагово.

Сначала я перемещу объявление переменной поближе к ее использованию.

```
function printOwing(invoice) {
  printBanner();

  // Выполнение вычислений
  let outstanding = 0;
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  recordDueDate(invoice);
  printDetails(invoice, outstanding);
}
```

Затем я копирую код, который хочу извлечь, в целевую функцию.

```
function printOwing(invoice) {
  printBanner();
  // Выполнение вычислений
  let outstanding = 0;
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }

  recordDueDate(invoice);
  printDetails(invoice, outstanding);
}

function calculateOutstanding(invoice) {
  let outstanding = 0;
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }
  return outstanding;
}
```

Поскольку я перенес объявление `outstanding` в извлеченный код, мне не нужно передавать его в качестве параметра. Переменная `outstanding` — единственная, присваиваемая в извлеченном коде, поэтому я могу вернуть ее из функции.

Следующее, что я должен сделать, — это заменить исходный код вызовом новой функции. Поскольку я возвращаю значение, мне нужно сохранить его в исходной переменной.

```
function printOwing(invoice) {
  printBanner();
  let outstanding = calculateOutstanding(invoice);
  recordDueDate(invoice);
  printDetails(invoice, outstanding);
}

function calculateOutstanding(invoice) {
  let outstanding = 0;
  for (const o of invoice.orders) {
    outstanding += o.amount;
  }
  return outstanding;
}
```

Прежде чем считать работу завершенной, я переименовываю возвращаемое значение в соответствии с моим обычным стилем кодирования.

```
function printOwing(invoice) {
  printBanner();
  const outstanding = calculateOutstanding(invoice);
  recordDueDate(invoice);
  printDetails(invoice, outstanding);
}

function calculateOutstanding(invoice) {
  let result = 0;
  for (const o of invoice.orders) {
    result += o.amount;
  }
  return result;
}
```

Я также пользуюсь возможностью сделать исходную переменную `outstanding` константной.

Вы можете спросить: "А что делать, если надо вернуть не одну, а несколько переменных?"

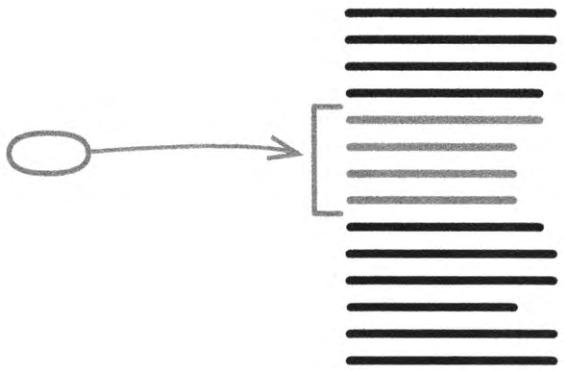
Есть ряд способов сделать это. Обычно лучше всего выбрать для извлечения другой код. Лично я предпочитаю, чтобы функция возвращала только одно значение, поэтому я попытался бы организовать возврат разных значений разными функциями. Если мне действительно необходимо извлечение функции с возвратом нескольких значений, я могу сформировать и вернуть запись — но обычно я предпочитаю переделать временные переменные. Для этого я бы предпочел применение рефакторингов *Замена временной переменной запросом* (с. 225) и *Расщепление переменной* (с. 285).

Это приводит к интересному вопросу, когда я извлекаю функции, которые планирую затем переместить в другой контекст, такой как верхний уровень. Я предпочитаю маленькие шаги, поэтому мой подход заключается в том, чтобы сначала извлечь код во вложенную функцию, а затем переместить эту вложенную функцию в ее новый контекст. Но сложность заключается в работе с переменными, и я не увижу этого, пока не начну работу. Это говорит о том, что хотя я могу извлечь код во вложенную функцию, стоит выполнить извлечение по крайней мере до уровня, совпадающего с уровнем исходной функции, и я смогу сразу сказать, имеет ли смысл извлеченный код.

Встраивание функции (Inline Function)

Бывший рефакторинг *Встраивание метода*

Обратный к рефакторингу *Извлечение функции* (с. 152)



```
function getRating(driver) {
    return moreThanFiveLateDeliveries(driver) ? 2 : 1;
}

function moreThanFiveLateDeliveries(driver) {
  return driver.numberOfLateDeliveries > 5;
}
```

```
function getRating(driver) {
  return (driver.numberOfLateDeliveries > 5) ? 2 : 1;
}
```

Мотивация

Данная книга учит использовать короткие функции с названиями, отражающими их предназначение, что дает более понятный и простой код. Но иногда встречаются функции, тела которых так же очевидны, как и названия, — либо изначально, либо после рефакторинга. В этом случае можно избавиться от функции. Косвенность может быть полезной, но излишняя косвенность мешает.

Еще одна ситуация, в которой можно применить указанный рефакторинг, — наличие группы функций, представляющейся плохо разделенной. Их все можно встроить в одну большую функцию, а затем выполнить извлечение функций другим способом.

Я обычно использую данный рефакторинг, когда нахожу в коде слишком много косвенности, и выясняю, что каждая функция просто выполняет делегирование другой функции, и во всем этом делегировании очень легко заблудиться. Косвенность оправданна далеко не всегда и далеко не в любых количествах. Встраивание позволяет собрать полезное и убрать ненужное.

Техника

- Убедитесь, что метод не является полиморфным.

  Если это метод класса, и имеются перекрывающие его подклассы — его нельзя встроить.
- Найдите все вызовы функции.
- Замените каждый вызов телом функции.
- Выполните компиляцию и тестирование после каждой замены.

  Встраивание не должно быть сделано все сразу. Если некоторые части встраивания оказываются сложными, их можно сделать постепенно.
- Удалите объявление функции.

Приведенное описание рефакторинга создает впечатление простоты, но в общем случае это не так. Я мог бы посвятить многие страницы обработке рекурсии, множественным точкам возврата, встраиванию в другие объекты при отсутствии функций доступа и т.п. Я не делаю этого потому, что при наличии таких сложностей данный рефакторинг лучше не применять.

Пример

В простейшем случае этот рефакторинг тривиален. Я начинаю с

```
function rating(aDriver) {
  return moreThanFiveLateDeliveries(aDriver) ? 2 : 1;
}
function moreThanFiveLateDeliveries(aDriver) {
  return aDriver.numberOfLateDeliveries > 5;
}
```

Я могу просто взять возвращаемое выражение вызываемой функции и вставить его в вызывающую функцию вместо вызова.

```
function rating(aDriver) {
  return aDriver.numberOfLateDeliveries > 5 ? 2 : 1;
}
```

Но все может быть немного более сложным и потребовать от меня больше работы. Рассмотрим небольшую вариацию предыдущего исходного кода.

```
function rating(aDriver) {
  return moreThanFiveLateDeliveries(aDriver) ? 2 : 1;
}
function moreThanFiveLateDeliveries(dvr) {
  return dvr.numberOfLateDeliveries > 5;
}
```

Почти то же самое, но теперь объявленный аргумент в `moreThanFiveLateDeliveries` отличается от имени передаваемого аргумента. Поэтому я должен немного подправить код при встраивании.

```
function rating(aDriver) {
  return aDriver.numberOfLateDeliveries > 5 ? 2 : 1;
}
```

Все может быть даже еще более сложным. Рассмотрим такой код.

```
function reportLines(aCustomer) {
  const lines = [];
  gatherCustomerData(lines, aCustomer);
  return lines;
}
function gatherCustomerData(out, aCustomer) {
  out.push(["name", aCustomer.name]);
  out.push(["location", aCustomer.location]);
}
```

Встраивание `collectCustomerData` в `reportLines` — не простая вырезка и вставка текста. Это не так уж сложно, и в большинстве случаев я делал бы все за один шаг, с небольшой подгонкой. Но сейчас я буду осторожным, выполняя перемещение по одной строке за раз. Так что начну с использования рефакторинга *Перенос инструкций в точку вызова* (с. 262) для первой строки.

```
function reportLines(aCustomer) {
  const lines = [];
  lines.push(["name", aCustomer.name]);
  gatherCustomerData(lines, aCustomer);
  return lines;
}
function gatherCustomerData(out, aCustomer) {
  out.push(["name", aCustomer.name]);
  out.push(["location", aCustomer.location]);
}
```

Затем продолжу работать с остальными строками, пока не завершу рефакторинг полностью.

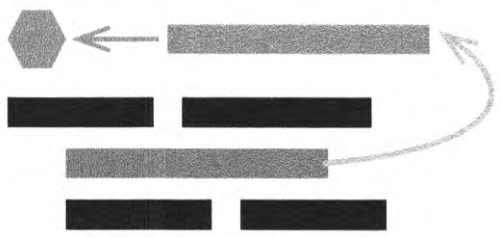
```
function reportLines(aCustomer) {
  const lines = [];
  lines.push(["name", aCustomer.name]);
  lines.push(["location", aCustomer.location]);
  return lines;
}
```

Суть в том, чтобы всегда быть готовым к меньшим шагам. Небольшие функции, которые я обычно пишу, позволяют мне выполнить данный рефакторинг за один раз. Но если я сталкиваюсь с осложнениями, то действую построчно. Даже

с одной строкой могут возникнуть проблемы; в таком случае я буду использовать более сложную механику рефакторинга *Перенос инструкций в точку вызова* (с. 262). Но если, чувствуя себя уверенно, я делаю что-то быстро, и мои тесты проваливаются, я предпочитаю вернуться к моему последнему "зеленому" коду и повторить рефакторинг с меньшими шагами (и легким огорчением).

Извлечение переменной (Extract Variable)

Бывший рефакторинг *Введение поясняющей переменной*
Обратный к рефакторингу *Встраивание переменной* (с. 169)



```
return order.quantity * order.itemPrice
  -Math.max(0,order.quantity-500)*order.itemPrice*0.05
  +Math.min(order.quantity*order.itemPrice*0.1,100);
```

```
const basePrice = order.quantity * order.itemPrice;
const quantityDiscount = Math.max(0,order.quantity-500)
                       * order.itemPrice * 0.05;
const shipping = Math.min(basePrice * 0.1, 100);
return basePrice - quantityDiscount + shipping;
```

Мотивация

Выражения могут быть очень сложными и трудными для чтения. В таких случаях имеет смысл с помощью локальных переменных превратить выражение в нечто, лучше поддающееся пониманию и управлению. В частности, локальные переменные дают возможность именовать часть более сложной части логики, что позволяет лучше понять цель происходящего.

Такие переменные также удобны для отладки, поскольку обеспечивают простоту просмотра в отладчике или при отладочном выводе.

Если я думаю о применении данного рефакторинга, значит, я хочу добавить имя к выражению в моем коде. Как только прихожу к такому решению, я размышляю также над контекстом этого имени. Если оно имеет смысл только в функции, над которой я работаю, то данный рефакторинг является хорошим выбором, но если оно имеет смысл в более широком контексте, то следует подумать о том, чтобы сделать имя доступным в этом более широком контексте, обычно — в функции. Если имя более широкодоступно, другой код может использовать это выражение без необходимости повторять само выражение, что приводит к уменьшению дублирования и лучшему изложению моих намерений.

Недостатком подъема имени в более широкий контекст являются дополнительные усилия. Если усилий требуется значительно больше, я, скорее всего, оставлю эту проблему на более поздний срок, когда смогу использовать рефакторинг *Замена временной переменной запросом* (с. 225). Но если сделать это легко, я предпочту выполнить подъем имени сразу же, чтобы оно стало доступно в коде тут же. Хорошим примером является работа в рамках класса, когда выполнить рефакторинг *Извлечение функции* (с. 152) очень легко.

Техника

- Убедитесь, что выражение, которое вы хотите извлечь, не имеет побочных действий.
- Объявите неизменяемую переменную. Присвойте ей копию выражения, которое вы хотите именовать.
- Замените исходное выражение новой переменной.
- Выполните тестирование.

Если выражение встречается более одного раза, замените каждое его вхождение переменной, выполняя тестирование после каждой замены.

Пример

Начну с простого вычисления

```
function price(order) {
  // Базовая цена - со скидкой и доставкой
  return order.quantity * order.itemPrice -
    Math.max(0,order.quantity-500)*order.itemPrice*0.05 +
    Math.min(order.quantity*order.itemPrice*0.1, 100);
}
```

Как бы просто это ни выглядело, я могу сделать так, чтобы понять код было еще легче. Во-первых, я осознаю, что базовая цена кратна количеству и цене единицы товара.

```
function price(order) {
  // Базовая цена - со скидкой и доставкой
  return order.quantity * order.itemPrice -
    Math.max(0, order.quantity-500)*order.itemPrice*0.05 +
    Math.min(order.quantity*order.itemPrice*0.1, 100);
}
```

Когда это понимание формируется у меня в голове, я помещаю его в код, создавая и именуя для него переменную.

```
function price(order) {
  // Базовая цена - со скидкой и доставкой
  const basePrice = order.quantity * order.itemPrice;
  return order.quantity * order.itemPrice -
    Math.max(0, order.quantity-500)*order.itemPrice*0.05 +
    Math.min(order.quantity*order.itemPrice*0.1, 100);
}
```

Конечно, простое объявление и инициализация переменной ничего не дает. Я должен использовать его, поэтому я заменяю выражение, которое использовал в качестве исходного.

```
function price(order) {
  // Базовая цена - со скидкой и доставкой
  const basePrice = order.quantity * order.itemPrice;
  return basePrice -
    Math.max(0, order.quantity-500)*order.itemPrice*0.05 +
    Math.min(order.quantity*order.itemPrice*0.1, 100);
}
```

Это же выражение применяется и позже, поэтому я могу заменить его переменной.

```
function price(order) {
  // Базовая цена - со скидкой и доставкой
  const basePrice = order.quantity * order.itemPrice;
  return basePrice -
    Math.max(0, order.quantity-500)*order.itemPrice*0.05 +
    Math.min(basePrice*0.1, 100);
}
```

Следующая строка представляет собой скидку, которую я тоже могу извлечь в переменную

```
function price(order) {
  // Базовая цена - со скидкой и доставкой
  const basePrice = order.quantity * order.itemPrice;
  const quantityDiscount =
    Math.max(0,order.quantity-500)*order.itemPrice*0.05;
  return basePrice -
         quantityDiscount +
         Math.min(basePrice * 0.1, 100);
}
```

Наконец, переходим к доставке. После этого я могу удалить комментарий, потому что он больше не говорит ничего такого, чего не говорит код.

```
function price(order) {
  const basePrice = order.quantity * order.itemPrice;
  const quantityDiscount =
    Math.max(0,order.quantity-500)*order.itemPrice*0.05;
  const shipping = Math.min(basePrice * 0.1, 100);
  return basePrice - quantityDiscount + shipping;
}
```

Пример с классом

Вот тот же самый код в контексте класса.

```
class Order {
  constructor(aRecord) {
    this._data = aRecord;
  }

  get quantity() {return this._data.quantity;}
  get itemPrice() {return this._data.itemPrice;}

  get price() {
    return this.quantity * this.itemPrice -
      Math.max(0, this.quantity-500)*this.itemPrice*0.05 +
      Math.min(this.quantity*this.itemPrice*0.1, 100);
  }
}
```

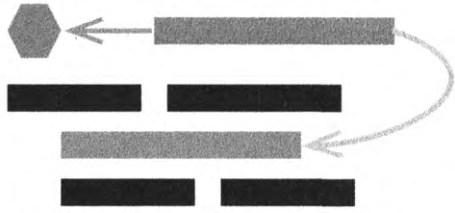
В этом случае я хочу извлечь те же имена, но понимаю, что имена относятся к `Order` в целом, а не только к вычислению цены. Поскольку они применяются ко всему заказу, я склонен извлекать имена как методы, а не как переменные.

```
class Order {
  constructor(aRecord) {
    this._data = aRecord;
  }
  get quantity() {return this._data.quantity;}
  get itemPrice() {return this._data.itemPrice;}

  get price() {
    return this.basePrice-this.quantityDiscount+this.shipping;
  }
  get basePrice() {return this.quantity*this.itemPrice;}
  get quantityDiscount() {
    return Math.max(0,this.quantity-500)*this.itemPrice*0.05;}
  get shipping() {return Math.min(this.basePrice * 0.1, 100);}
}
```

Это одно из больших преимуществ объектов — они дают достаточный контекст для совместного использования логики с другими фрагментами логики и данных. Для чего-то столь же простого, как приведенный пример, это не имеет большого значения, но в случае большего класса оказывается очень полезным вызов общих фрагментов поведения как их собственных абстракций со своими собственными именами, позволяющими обращаться к ним при работе с объектом.

Встраивание переменной (Inline Variable)

Бывший рефакторинг *Встраивание временной переменной*
Обратный к рефакторингу *Извлечение переменной* (с. 165)



```
let basePrice = anOrder.basePrice;
return (basePrice > 1000);
```

```
return anOrder.basePrice > 1000;
```

Мотивация

Переменные предоставляют имена для выражений внутри функции, и поэтому они обычно являются Полезными Вещами. Но иногда в действительности имя не говорит больше, чем само выражение. В других случаях вы можете обнаружить, что переменная мешает рефакторингу соседнего кода. В этих случаях может быть полезно встроить переменную.

Техника

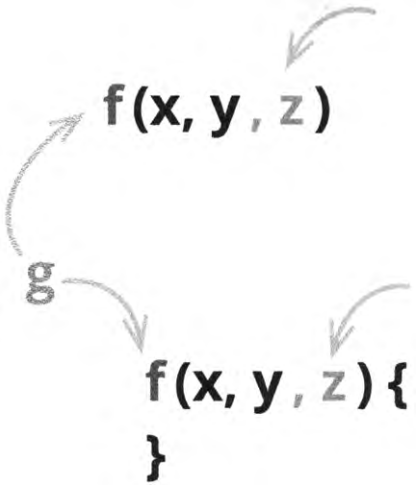
- Убедитесь, что справа от знака присваивания нет никаких побочных действий.
- Если переменная не объявлена как неизменяемая — объявите ее таковой и выполните тестирование.

  Это проверка того, что присваивание выполняется только один раз.

- Найдите первое обращение к переменной и замените ее правой частью присваивания.
- Выполните тестирование.
- Повторите замену обращений к переменной, пока не замените их все.
- Удалите объявление и присваивание переменной.
- Выполните тестирование.

Изменение объявления функции (Change Function Declaration)

Он же: *Переименование функции*
Бывший рефакторинг *Переименование метода*
Бывший рефакторинг *Добавление параметра*
Бывший рефакторинг *Удаление параметра*
Он же: *Изменение сигнатуры*




```
function circum(radius) {...}
```

```
function circumference(radius) {...}
```

Мотивация

Функции представляют собой основной способ разделения программы на части. Объявления функций показывают, как эти части сочетаются одна с другой — фактически они представляют собой соединения в наших программных системах. И, как и в случае любой конструкции, многое зависит от этих соединений. Хорошие соединения позволяют легко добавлять новые компоненты в систему, а плохие являются постоянным источником проблем, затрудняя понимание того, что делает программа, и как ее можно модифицировать, когда изменяются потребности. К счастью, "мягкость" программного обеспечения[1] позволяет менять эти соединения — при условии, что это делается осторожно.

Наиболее важным элементом такого соединения является имя функции. Хорошее имя позволяет понять, что делает функция, не видя кода, который ее реализует. Однако найти хорошие имена сложно, и лично у меня с первого раза это получается очень редко. Когда я нахожу имя, которое меня смущает, первое мое желание — оставить его как есть: в конце концов, это всего лишь имя. Это работа злого демона *Обфускатиса*[2]; ради спасения души программы я никогда не должен слушать его. Если я вижу функцию с неправильным именем, я обязательно должен изменить его, подобрав лучшее имя. Таким образом, в следующий раз, когда я буду работать с этим кодом, мне не придется *вновь* выяснять, что в нем происходит. (Зачастую хороший способ улучшить имя — написать комментарий, описывающий назначение функции, а затем превратить этот комментарий в ее имя.)

Аналогичная логика применима и к параметрам функции. Параметры функции определяют, как функция вписывается в остальной мир, устанавливая контекст, в котором можно использовать функцию. Если у меня есть функция для форматирования телефонных номеров людей, и если эта функция принимает в качестве аргумента объект типа `person`, то я не могу использовать ее для форматирования телефонного номера компании. Если я заменю параметр `person` номером телефона, то код для форматирования номера будет более полезным.

Помимо увеличения применимости функции, можно также уменьшить степень связности, изменяя подключение одних модулей к другим. Логика форматирования номера телефона может находиться в модуле, который ничего не знает о людях. Снижение количества модулей, которым нужно знать о других модулях, помогает уменьшить количество информации, которое я должен вкладывать в мозг при внесении тех или иных изменений.

Выбор правильных параметров не является набором простых правил. У меня может быть простая функция, позволяющая определить, просрочен ли платеж,

---

[1] Непереводимая игра слов: программное обеспечение — software; soft — мягкий. — *Примеч. пер.*

[2] Obfuscate (англ.) — затемнять, запутывать. — *Примеч. пер.*

путем проверки, не старше ли он 30 дней. Должен ли параметр этой функции быть объектом платежа или датой платежа? Использование объекта платежа связывает функцию с интерфейсом этого объекта. Но если я использую платеж, то при развитии логики я смогу легко получить доступ и к другим свойствам платежа без необходимости изменять каждый фрагмент кода программы, вызывающий эту функцию, — по сути, увеличивая степень инкапсуляции функции.

Единственный правильный ответ на этот вопрос состоит в том, что правильного ответа нет — особенно с течением времени. Поэтому я считаю, что знакомство с рефакторингом *Изменение объявления функции* позволяет коду развиваться параллельно с развитием понимания, какими должны быть упоминавшиеся выше соединения в программной системе.

Обычно я использую основное имя рефакторинга, только когда ссылаюсь на него из других мест в книге. Но поскольку переименование является очень важным частным случаем применения рефакторинга *Изменение объявления функции*, в случае переименования я буду ссылаться на этот рефакторинг как на *Переименование функции*, чтобы было понятнее, что именно я делаю. Но и когда я просто переименовываю функцию, и когда манипулирую ее параметрами — я использую одну и ту же технику.

Техника

В большинстве рефакторингов этой книги я представляю только один набор действий. Я поступаю так не потому, что существует только один правильный путь выполнения работы, а потому, что, как правило, один и тот же набор действий будет достаточно хорошо работать для большинства случаев. Однако данный рефакторинг является исключением. Простая техника часто оказывается эффективной, но встречается немало ситуаций, когда больший смысл имеет более постепенная миграция. Таким образом, при выполнении этого рефакторинга я смотрю на необходимое изменение и спрашиваю себя, смогу ли я легко изменить объявление и все вызовы функции за один раз. Если да — я следую простой технике. Техника в стиле миграции позволяет мне изменять вызовы функции более постепенно — что важно, если у меня их много, если их сложно искать, если функция представляет собой полиморфный метод, или если у меня предполагаются более сложные изменения объявления.

Простая техника

- Если вы удаляете параметр, убедитесь, что к нему нет обращений в теле функции.
- Измените объявление метода на желаемое.

- Найдите все обращения к старому объявлению функции и измените их на обращения к новому объявлению.
- Выполните тестирование.

Часто лучше разделять изменения, поэтому, если вы хотите изменить имя и добавить параметр, выполните это как отдельные шаги. (В любом случае, если у вас возникнут проблемы, вернитесь и используйте вместо этого технику миграции.)

Техника миграции

- При необходимости выполните рефакторинг тела функции, чтобы упростить выполнение следующего шага извлечения.
- Выполните рефакторинг *Извлечение функции* (с. 152) над телом функции для создания новой функции.

  Если новая функция будет иметь то же имя, что и старая, — дайте новой функции временное имя, которое будет просто искать.
- Если извлеченная функция нуждается в дополнительных параметрах, используйте простую механику для их добавления.
- Выполните тестирование.
- Примените рефакторинг *Встраивание функции* (с. 161) к старой функции.
- Если вы использовали временное имя, примените рефакторинг *Изменение объявления функции* (с. 170) еще раз для восстановления исходного имени.
- Выполните тестирование.

Если вы изменяете метод в классе с полиморфизмом, для каждого связывания будет необходимо добавить косвенность. Если метод полиморфен в иерархии одного класса, вам нужен только метод передачи в суперклассе. Если полиморфизм не связан с суперклассом, вам понадобятся методы передачи для каждого класса реализации.

Если вы выполняете рефакторинг опубликованного API, то можете приостановить рефакторинг после создания новой функции. Во время этой паузы использовать исходную функцию не рекомендуется. Подождите, пока все клиенты не перейдут на новую функцию. Исходное объявление функции можно удалять, лишь убедившись, что все клиенты старой функции перешли на новую.

Пример: переименование функции (простая техника)

Рассмотрим следующую функцию с чрезмерно сокращенным именем.

```
function circum(radius) {
  return 2 * Math.PI * radius;
}
```

Я хочу изменить его на что-то более разумное. Я начинаю с изменения объявления.

```
function circumference(radius) {
  return 2 * Math.PI * radius;
}
```

Затем я нахожу все вызовы функции `circum` и изменяю вызываемое имя на `circumference`.

Различные языковые среды по-разному помогают в решении задачи поиска всех обращений к старой функции. Статическая типизация и хорошая интегрированная среда разработки обеспечивают хорошую помощь, обычно позволяя мне переименовывать функции автоматически с мизерной вероятностью ошибки. Без статической типизации дело может быть более сложным; даже у хороших инструментов поиска может быть много ложных срабатываний.

Я использую тот же подход для добавления или удаления параметров: найти все вызовы, изменить объявление и изменить вызовы. Часто лучше сделать это в виде отдельных шагов — так что если я одновременно переименовываю функцию и добавляю параметр, то сначала выполняю переименование, тестирую, затем добавляю параметр и снова тестирую.

Недостаток этого простого способа выполнения рефакторинга состоит в том, что я должен обновлять все вызовы и объявление (или все объявления в случае полиморфности) одновременно. Если их всего лишь несколько, или если у меня есть приличные инструменты автоматического рефакторинга, это разумно. Но если таких мест много, это может быть слишком сложно. Другая проблема заключается в том, что имена не являются уникальными. Например, я хочу переименовать метод `changeAddress` в классе персоны, но имеется одноименный метод, который я не хочу изменять, в классе договора страхования. Чем сложнее изменение, тем меньше я хочу выполнять его за один раз. Когда возникает такая проблема, я использую технику миграции. Точно так же, если я использую простую технику и что-то идет не так, я возвращаю код в последнее известное исправное состояние и пытаюсь воспользоваться техникой миграции.

Пример: переименование функции (техника миграции)

Я вновь хочу изменить функцию с чрезмерно сокращенным именем:

```
function circum(radius) {
  return 2 * Math.PI * radius;
}
```

Для выполнения этого рефакторинга с использованием техники миграции я начинаю с применения рефакторинга *Извлечение функции* (с. 152) ко всему телу функции.

```
function circum(radius) {
  return circumference(radius);
}
function circumference(radius) {
  return 2 * Math.PI * radius;
}
```

Я выполняю тестирование, затем применяю рефакторинг *Встраивание функции* (с. 161) к старой функции. Я нахожу все вызовы старой функции и заменяю каждый вызовом новой. Я могу выполнять тестирование после каждого внесенного изменения, что позволяет мне делать их по одному. Выполнив все замены, я удаляю старую функцию.

В большинстве рефакторингов я вношу изменения в код, который можно менять, но этот рефакторинг может быть полезен и в случае опубликованного API, используемого кодом, который я не могу изменить самостоятельно. Я могу приостановить рефакторинг после создания `circumference` и, если это возможно, пометить функцию `circum` как устаревшую. Затем я жду, пока клиенты, работающие с `circum`, не перейдут с этой функции на применение `circumference`; как только они это сделают, я смогу удалить старую функцию. Даже если я никогда не смогу достичь счастливого момента удаления `circum`, по крайней мере у меня будет лучшее имя для нового кода.

Пример: добавление параметра

В некоторой программе для управления обычной книжной библиотекой у меня есть класс книги, в котором имеется возможность забронировать книгу для клиента.

```
class Book...
addReservation(customer) {
  this._reservations.push(customer);
}
```

Для бронирования мне нужно поддерживать очередь с приоритетами. Таким образом, мне нужен дополнительный параметр в функции `addReservation`, чтобы указать, должно ли бронирование выполняться в обычной очереди или в очереди с высоким приоритетом. Если я могу легко найти и изменить все вызовы функции, то я могу просто выполнить это изменение, но если нет — я могу использовать показанную ниже технику миграции.

Я начинаю с использования рефакторинга *Извлечение функции* (с. 152) в теле `addReservation` для создания новой функции. Хотя в конечном итоге она будет называться `addReservation`, новые и старые функции с одним и тем же именем не могут сосуществовать одновременно. Поэтому я использую временное имя, которое будет легко найти позже.

```
class Book...
addReservation(customer) {
  this.zz_addReservation(customer);
}
zz_addReservation(customer) {
  this._reservations.push(customer);
}
```

Затем я добавляю параметр в новое объявление и вызов функции (используя очень простую технику).

```
class Book...
addReservation(customer) {
  this.zz_addReservation(customer, false);
}
zz_addReservation(customer, isPriority) {
  this._reservations.push(customer);
}
```

При работе с JavaScript перед тем, как изменить любую точку вызова, я предпочитаю применять рефакторинг *Введение утверждения* (с. 346), чтобы проверить, используется ли в этой точке новый параметр.

```
class Book...
zz_addReservation(customer, isPriority) {
  assert(isPriority === true || isPriority === false);
  this._reservations.push(customer);
}
```

Теперь, когда я вношу изменение в точке вызова, если я ошибусь и пропущу новый параметр, это утверждение поможет мне обнаружить ошибку. Из многолетнего опыта работы я знаю, что более склонных к ошибкам программистов, чем я, найти не так-то просто.

Теперь я могу начать изменять точки вызова, используя рефакторинг *Встраивание функции* (с. 161) для исходной функции. Это позволяет мне менять по одной точке вызова за раз.

Затем я переименовываю новую функцию, давая ей имя исходной функции. Обычно для этого отлично подходит простая техника, но если нужно — я могу использовать и подход с миграцией.

Пример: замена параметра одним из его свойств

Рассмотренные до сих пор примеры состояли в простом изменении имени и добавлении нового параметра, но с применением миграции данный рефакторинг может довольно аккуратно обрабатывать и более сложные случаи. Рассмотрим немного более сложный пример.

У меня есть функция, которая определяет, находится ли клиент в Новой Англии[3].

```
function inNewEngland(aCustomer) {
  return ["MA", "CT", "ME", "VT", "NH", "RI"]
         .includes(aCustomer.address.state);
}
```

Вот пример одного из ее вызовов:

```
Точка вызова...
const newEnglanders = someCustomers.filter(c=>inNewEngland(c));
```

Функция `inNewEngland` для определения того, находится ли клиент в Новой Англии, использует его штат. Я хотел бы выполнить рефакторинг `inNewEngland`, чтобы она принимала в качестве параметра код штата, что, удаляя зависимость от клиента, сделало бы ее пригодной для использования в большем количестве контекстов.

Мой первый обычный шаг при использовании рассматриваемого рефакторинга *Изменение объявления функции* — применение рефакторинга *Извлечение функции* (с. 152), но в данном случае выполнение рефакторинга можно упростить, немного реорганизовав тело функции. Я воспользуюсь рефакторингом *Извлечение переменной* (с. 165) для моего нового параметра.

```
function inNewEngland(aCustomer) {
  const stateCode = aCustomer.address.state;
  return ["MA","CT","ME","VT","NH","RI"].includes(stateCode);
}
```

Теперь я применяю рефакторинг *Извлечение функции* (с. 152) для создания новой функции.

```
function inNewEngland(aCustomer) {
  const stateCode = aCustomer.address.state;
  return xxNEWinNewEngland(stateCode);
}
function xxNEWinNewEngland(stateCode) {
  return ["MA","CT","ME","VT","NH","RI"].includes(stateCode);
}
```

Даю функции имя, которое позже будет очень легко автоматически заменить именем исходной функции. (Какого-либо стандарта для временных имен у меня нет.)

Применяю рефакторинг *Встраивание переменной* (с. 169) ко входному параметру исходной функции.

```
function inNewEngland(aCustomer) {
  return xxNEWinNewEngland(aCustomer.address.state);
}
```

[3] Регион на северо-востоке США, включающий штаты Мэн, Вермонт, Нью-Гемпшир, Массачусетс, Коннектикут и Род-Айленд. — *Примеч. пер.*

Использую рефакторинг *Встраивание функции* (с. 161), чтобы эффективно заменить вызов старой функции вызовом новой. Я могу выполнить эту замену по одной функции.

```
Точка вызова...
const newEnglanders =
  someCustomers.filter(c=>xxNEWinNewEngland(c.address.state));
```

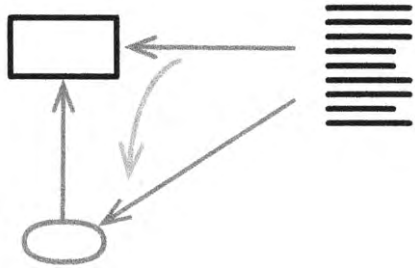
Встроив старую функцию в каждую точку вызова, я вновь использую наш текущий рефакторинг *Изменение объявления функции* для изменения имени новой функции на исходное.

```
Точка вызова...
const newEnglanders =
  someCustomers.filter(c => inNewEngland(c.address.state));

Верхний уровень...
function inNewEngland(stateCode) {
  return ["MA", "CT", "ME", "VT", "NH", "RI"].includes(stateCode);
}
```

Инструменты автоматизированного рефакторинга делают технику миграции менее полезной и более эффективной. Они делают ее менее полезной, потому что обрабатывают даже сложные переименования и изменения параметров безопаснее, поэтому мне не нужно прибегать к миграции так часто, как без такой автоматизированной поддержки. Однако в таких случаях, как данный пример, когда инструментарий не в состоянии выполнить весь рефакторинг, они делают его намного проще, поскольку с помощью инструментария можно быстрее и безопаснее выполнять ключевые операции извлечения и встраивания.

Инкапсуляция переменной (Encapsulate Variable)

Бывший рефакторинг *Самоинкапсуляция поля*
Бывший рефакторинг *Инкапсуляция поля*



```
let defaultOwner = {firstName:"Martin",lastName:"Fowler"};
```

```
let defaultOwnerData = {firstName:"Martin",lastName:"Fowler"};
export function defaultOwner() {return defaultOwnerData;}
export function setDefaultOwner(arg) {defaultOwnerData=arg;}
```

Мотивация

Рефакторинг работает с элементами наших программ. Работать с данными менее удобно, чем с функциями. Поскольку использование функции обычно означает ее вызов, я могу легко переименовать или переместить функцию, сохранив при этом старую версию функции как функцию передачи (мой старый код в результате вызывает старую функцию, которая просто вызывает новую функцию). Обычно эта функция пересылки поддерживается недолго, но она упрощает рефакторинг.

Данные более неудобны, потому что поступить так же с ними не получается. Если я перемещаю данные, то, чтобы код работал, мне нужно изменить все обращения к данным за один шаг. Для данных с очень небольшой областью видимости, таких как временные переменные в небольшой функции, это не проблема. Но с ростом масштабов увеличивается и сложность, и именно поэтому глобальные данные являются такой головной болью.

Так что, если я хочу переместить данные с широкой областью видимости, зачастую наилучший способ сделать это — сначала инкапсулировать их, перенаправив все обращения к ним через функции. Таким образом я превращаю трудную задачу реорганизации данных в более простую задачу реорганизации функций.

Инкапсуляция данных полезна и для других целей. Она обеспечивает четкую точку мониторинга изменений и использования данных. Я могу легко добавить проверку корректности или косвенную логику при обновлении данных. У меня есть привычка делать все изменяемые данные инкапсулированными и обращаться к ним только через функции, если их область действия превышает единственную функцию. Чем больше область видимости данных, тем важнее их инкапсуляция. Мой подход к старому коду заключается в том, что всякий раз, когда мне нужно изменить или добавить новое обращение к такой переменной, я пользуюсь этим для ее инкапсуляции. Таким образом я предотвращаю усиление связывания с часто используемыми данными.

Именно этот принцип заставляет объектно-ориентированный подход уделять большое внимание закрытости данных объекта. Всякий раз, когда я вижу открытое поле, я рассматриваю возможность использования данного рефакторинга (который в этом случае называется *Инкапсуляцией поля*), чтобы уменьшить его

видимость. Некоторые идут дальше и утверждают, что даже внутренние обращения к полям внутри класса должны выполняться через функции доступа — подход, известный как *самоинкапсуляция*. В общем случае я считаю самоинкапсуляцию чрезмерной — если класс настолько велик, что нужна самоинкапсуляция его полей, его все равно нужно разбивать на меньшие части. Но самоинкапсуляция поля — полезный шаг перед разделением класса.

Поддержка инкапсуляции данных гораздо менее важна для неизменяемых данных. Когда данные не меняются, мне не нужно место для размещения их проверки или других действий при их обновлениях. Я могу свободно копировать данные, а не перемещать их, поэтому мне не нужно менять обращения к ним из старых местоположений, и при этом я не беспокоюсь о том, какие части кода получают устаревшие данные. Неизменяемость является мощным средством предохранения.

Техника

- Создайте инкапсулирующие функции получения и установки значений поля.
- Выполните статические проверки.
- Для каждого обращения к переменной замените его вызовом соответствующей инкапсулирующей функции. Выполните тестирование после каждой замены.
- Ограничьте видимость переменной.

  Иногда невозможно предотвратить доступ к переменной. В таком случае может быть полезно обнаружить любые оставшиеся обращения, переименовав переменную и выполняя тестирование.
- Выполните тестирование.
- Если значение переменной представляет собой запись, рассмотрите применение рефакторинга *Инкапсуляция записи* (с. 208).

Пример

Рассмотрим некоторые полезные данные, хранящиеся в глобальной переменной.

```
let defaultOwner = {firstName: "Martin", lastName: "Fowler"};
```

Как и к любым данным, обращение к ним выполняется с помощью кода наподобие следующего:

```
spaceship.owner = defaultOwner;
```

а обновление — с помощью такого:

```
defaultOwner = {firstName: "Rebecca", lastName: "Parsons"};
```

Чтобы выполнить базовую инкапсуляцию, я начинаю с определения функций для чтения и записи данных.

```
function getDefaultOwner() {return defaultOwner;}
function setDefaultOwner(arg) {defaultOwner = arg;}
```

Затем я начинаю работать с обращениями к `defaultOwner`. Видя обращение, я заменяю его вызовом функции получения значения переменной.

```
spaceship.owner = getDefaultOwner();
```

Когда я встречаю присваивание, я заменяю его вызовом функции установки значения переменной.

```
setDefaultOwner({firstName: "Rebecca", lastName: "Parsons"});
```

После каждой замены я выполняю тестирование.

Закончив работу со всеми обращениями, я ограничу видимость переменной, чтобы убедиться, что не пропустил никаких обращений, и гарантировать, что будущие изменения в коде не смогут получить непосредственный доступ к переменной. В JavaScript я могу сделать это, переместив как переменную, так и методы доступа в их собственный файл и экспортировав из него только методы доступа.

```
defaultOwner.js...
let defaultOwner = {firstName: "Martin", lastName: "Fowler"};
export function getDefaultOwner() {return defaultOwner;}
export function setDefaultOwner(arg) {defaultOwner = arg;}
```

Не имея возможности ограничить доступ к переменной, полезно переименовать переменную и выполнить повторную проверку. Это не помешает будущему непосредственному доступу, но именование переменной каким-то значимым и неудобным именем, таким как `__privateOnly_defaultOwner`, может заставить будущего разработчика задуматься.

Мне не нравится использование префиксов `get` в функциях доступа, поэтому я их переименую.

```
defaultOwner.js...
let defaultOwnerData = {firstName: "Martin", lastName: "Fowler"};
export function ~~get~~defaultOwner() {return defaultOwnerData;}
export function setDefaultOwner(arg) {defaultOwnerData = arg;}
```

Общепринятым соглашением в JavaScript является одинаковое именование функции получения и установки, и их различие по наличию аргумента. Я называю этот подход "Перегруженными функциями доступа" [28] и очень его не люблю. Поэтому, хотя мне и не нравится префикс `get`, префикс `set` я сохраняю.

Инкапсуляция значения

Базовый рефакторинг, изложенный здесь, инкапсулирует обращение к некоторой структуре данных, что позволяет мне управлять обращениями к ней и присваиваниями — но не управляет изменениями этой структуры.

```
const owner1 = defaultOwner();
assert.equal("Fowler", owner1.lastName, "when set");
const owner2 = defaultOwner();
owner2.lastName = "Parsons";
assert.equal("Parsons", owner1.lastName,
             "after change owner2");    // Все в порядке?
```

Базовый рефакторинг инкапсулирует обращение к элементу данных. Во многих случаях это все, что в данный момент я хочу сделать. Но зачастую мне хочется углубить инкапсуляцию, чтобы контролировать не только изменения переменной, но и ее содержимого.

Для этого у меня есть пара вариантов. Самый простой способ — предотвратить любые изменения значения. Мой любимый способ сделать это — изменить функцию получения данных таким образом, чтобы она возвращала копию данных.

```
defaultOwner.js...
let defaultOwnerData = {firstName:"Martin",lastName:"Fowler"};
export function defaultOwner() {
  return Object.assign({}, defaultOwnerData);}
export function setDefaultOwner(arg) {defaultOwnerData = arg;}
```

Особенно часто я применяю этот подход для списков. Если я верну копию данных, любые клиенты, работающие с ней, могут ее изменить, но это изменение не отразится на совместно используемых данных. Однако с использованием копий нужно быть осторожным: некоторый код может ожидать изменения совместно используемых данных. Если это так, я полагаюсь в обнаружении проблемы на свои тесты. Альтернативой является предотвращение изменений, которого можно достичь применением рефакторинга *Инкапсуляция записи* (с. 208).

```
let defaultOwnerData = {firstName: "Martin", lastName: "Fowler"};
export function defaultOwner(){return new Person(defaultOwnerData);}
export function setDefaultOwner(arg) {defaultOwnerData = arg;}

class Person {
  constructor(data) {
    this._lastName = data.lastName;
    this._firstName = data.firstName
  }
  get lastName() {return this._lastName;}
  get firstName() {return this._firstName;}
  // И т.д. для других свойств
```

Теперь любая попытка присваивания нового значения свойству владельца по умолчанию будет вести к ошибке. Разные языки используют различные методы обнаружения или предотвращения подобных изменений, поэтому я рассмотрю и другие варианты (в зависимости от языка).

Обнаружение и предотвращение таких изменений часто имеет смысл в качестве временной меры. Я могу либо удалить изменения, либо предоставить подходящие изменяющие функции. Затем, как только работа с ними будет завершена, я могу изменить метод доступа так, чтобы он возвращал копию.

До сих пор я говорил о копировании при получении данных, но, возможно, стоит сделать копию и в функции установки значения. Это зависит от того, откуда берутся данные и требуется ли поддерживать ссылку для отражения любых изменений в исходных данных. Если такая ссылка не нужна, копирование предотвращает неприятности из-за изменений исходных данных. В большинстве случаев копирование может быть излишним, но, с одной стороны, копирование в таких случаях обычно оказывает незначительное влияние на производительность; с другой стороны, если его не выполнить — есть риск долгой и трудной отладки в будущем.

Помните, что вышеупомянутое копирование и класс-оболочка работают в структуре записи только на один уровень глубже. Для более глубоких уровней требуется больше уровней копирования или оболочек объектов.

Как видите, несмотря на свою ценность, инкапсуляция данных оказывается зачастую весьма непростой. Что именно инкапсулировать (и как это делать), зависит от способа использования данных и планируемых изменений. Но чем шире она используется, тем больше внимания следует уделить правильному ее выполнению.

Переименование переменной (Rename Variable)


```
let a = height * width;
```

```
let area = height * width;
```

Мотивация

Правильное именование — это сердце правильного программирования. При правильном названии переменные могут помочь понять намерения программиста. Но лично я часто ошибаюсь в именах своих переменных — иногда потому, что недостаточно тщательно их продумываю, иногда потому, что мое понимание проблемы улучшается по мере работы над ней, а иногда потому, что сама цель программы меняется по мере того, как меняются потребности ее пользователей.

Важность имени существенно зависит от того, насколько широко оно используется. Переменная, используемая в однострочном лямбда-выражении, обычно проста для понимания — и в этом случае я часто использую одну букву, поскольку назначение переменной очевидно из ее контекста. Параметры коротких функций могут быть краткими по той же самой причине, хотя в динамически типизированном языке, таком как JavaScript, я предпочитаю добавлять в имя переменной ее тип (отсюда и имена параметров, такие как `aCustomer`).

Постоянные поля, которые хранятся дольше одного вызова функции, требуют более тщательного именования. Это то место, которому следует уделить особое внимание.

Техника

- Если переменная широко используется — рассмотрите применение рефакторинга *Инкапсуляция переменной* (с. 178).
- Найдите все обращения к переменной и измените их.

  Если имеются обращения из другой кодовой базы, такая переменная является открытой опубликованной переменной, и вы не можете применить к ней данный рефакторинг.

  Если переменная не изменяется — вы можете скопировать ее в переменную с новым именем, а затем выполнить изменения постепенно, с тестированием после внесения каждого из них.
- Выполните тестирование.

Пример

Самый простой случай переименования переменной — когда она локальна в отдельной функции, т.е. является временной переменной или аргументом. Эта ситуация слишком тривиальна даже для примера: я просто нахожу каждое обращение к переменной и меняю его. Завершив переименование, я выполняю тестирование, чтобы убедиться, что ничего не испортил.

Проблемы возникают, когда переменная имеет более широкую область видимости, чем единственная функция. В кодовой базе к ней может быть много обращений:

```
let tpHd = "untitled";
```

Некоторые обращения получают ее значения:

```
result += `<h1>${tpHd}</h1>`;
```

Другие обращения изменяют ее значение:

```
tpHd = obj['articleTitle'];
```

Моя обычная реакция — применение рефакторинга *Инкапсуляция переменной* (с. 178).

```
result += `<h1>${title()}</h1>`;

setTitle(obj['articleTitle']);

function title() {return tpHd;}
function setTitle(arg) {tpHd = arg;}
```

Теперь я могу переименовать переменную.

```
let _title = "untitled";

function title() {return _title;}
function setTitle(arg) {_title = arg;}
```

Я мог бы продолжить встраивать функции-обертки, чтобы во всех вызовах переменная использовалась непосредственно. Но я редко поступаю таким образом. Если переменная используется достаточно широко, чтобы я почувствовал необходимость ее инкапсуляции для того, чтобы изменить ее имя, — такую переменную стоит оставить инкапсулированной в функции на будущее.

В тех случаях, когда я собирался выполнить встраивание, я вызывал функцию доступа `getTitle` и не использовал подчеркивание в имени переменной при переименовании.

Переименование константы

Если я переименовываю константу (или что-то, что для клиентов действует как константа), я могу избежать инкапсуляции и выполнять переименование постепенно путем копирования. Если исходное объявление выглядит как

```
const cpyNm = "Acme Gooseberries";
```

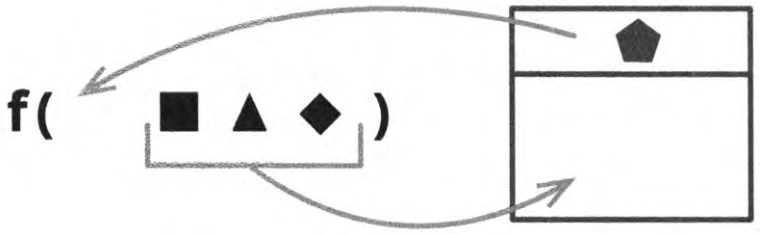
то я могу начать переименование с создания копии:

```
const companyName = "Acme Gooseberries";
const cpyNm = companyName;
```

При использовании копирования можно постепенно менять обращения со старого имени на новое. По завершении работы копия удаляется. Я предпочитаю объявлять новое имя и копировать его в старое, если это облегчает удаление старого имени.

Этот подход работает как для констант, так и для переменных, которые доступны клиентам только для чтения (например, экспортируемая переменная в JavaScript).

Введение объекта параметра (Introduce Parameter Object)




```
function amountInvoiced(startDate, endDate) {...}
function amountReceived(startDate, endDate) {...}
function amountOverdue(startDate, endDate) {...}
```

```
function amountInvoiced(aDateRange) {...}
function amountReceived(aDateRange) {...}
function amountOverdue(aDateRange) {...}
```

Мотивация

Я часто встречаю группы элементов данных, которые регулярно появляются вместе от функции к функции. Такая группа представляет собой связанный набор данных, и мне нравится заменять его единой структурой данных.

Преимуществом группировки данных в структуру является то, что она делает явной связь между этими данными. Это уменьшает размер списков параметров любой функции, которая использует новую структуру, и способствует согласованности, поскольку все функции, применяющие структуру, будут использовать одинаковые имена для доступа к ее элементам.

Но настоящая сила этого рефакторинга заключается в том, что он обеспечивает возможность более глубоких изменений кода. Идентифицируя эти новые структуры, я могу переориентировать программу на их использование. Я создам функции, которые собирают воедино общее поведение над этими данными — либо

как набор общих функций, либо как класс, который объединяет с этими функциями структуру данных. Данный процесс может изменить концептуальную картину кода, предоставив эти структуры в качестве новых абстракций, значительно упрощающих понимание предметной области. Все это может иметь удивительно мощное влияние на программу, но все это будет невозможно, если не использовать рассматриваемый в этом разделе рефакторинг *Введение объекта параметра*, чтобы начать процесс.

Техника

- Если подходящей структуры еще нет — создайте таковую.

  Я стараюсь использовать класс, так как в дальнейшем это облегчает группирование поведения. Обычно я предпочитаю, чтобы эти структуры были объектами-значениями [39].
- Выполните тестирование.
- Используйте рефакторинг *Изменение объявления функции* (с. 170) для добавления параметра в новую структуру.
- Выполните тестирование.

Настройте каждую точку вызова, чтобы функции передавался корректный экземпляр новой структуры. После каждого изменения выполняйте тестирование.

Для каждого элемента новой структуры замените использование исходного параметра элементом структуры. Удалите параметр. Выполните тестирование.

Пример

Я начну с некоторого кода, который просматривает набор показаний температуры и определяет, не выходят ли какие-либо из них за пределы рабочего диапазона. Вот как выглядят данные:

```
const station = { name: "ZB1",
                  readings: [
                    {temp: 47, time: "2016-11-10 09:10"},
                    {temp: 53, time: "2016-11-10 09:20"},
                    {temp: 58, time: "2016-11-10 09:30"},
                    {temp: 53, time: "2016-11-10 09:40"},
                    {temp: 51, time: "2016-11-10 09:50"},
                  ]
                };
```

У меня есть функция для поиска данных, выходящих за пределы диапазона температур.

```
function readingsOutsideRange(station, min, max) {
  return station.readings.filter(r => r.temp<min||r.temp>max);
}
```

Она может быть вызвана из некоторого кода следующим образом:

```
Точка вызова...
alerts = readingsOutsideRange(station,
                              operatingPlan.temperatureFloor,
                              operatingPlan.temperatureCeiling);
```

Обратите внимание, что вызывающий код извлекает два элемента данных из другого объекта и передает эту пару в функцию `readingsOutsideRange`. По сравнению с `readingsOutsideRange` рабочий план (`operatingPlan`) использует разные имена для обозначения начала и конца диапазона. Такого рода диапазон является распространенным случаем, когда два отдельных элемента данных лучше объединять в один объект. Я начну с объявления класса для объединенных данных.

```
class NumberRange {
  constructor(min, max) {
    this._data = {min: min, max: max};
  }
  get min() {return this._data.min;}
  get max() {return this._data.max;}
}
```

Я объявляю класс, а не просто использую базовый объект JavaScript, потому что обычно рассматриваю этот рефакторинг как первый шаг к перемещению поведения во вновь создаваемый объект (а именно в этом и состоит смысл класса). Я не предоставляю какие-либо методы обновления для нового класса, так как, вероятно, сделаю его объектом-значением [39]. В большинстве случаев при выполнении этого рефакторинга я создаю именно объекты-значения.

Затем я использую рефакторинг *Изменение объявления функции* (с. 170), чтобы добавить новый объект в качестве параметра функции `readingsOutsideRange`.

```
function readingsOutsideRange(station, min, max, range) {
  return station.readings.filter(r => r.temp<min||r.temp>max);
}
```

В JavaScript я могу оставить вызывающий код без изменений, но в других языках программирования я должен добавить нулевой аргумент — что-то вроде

```
Точка вызова...
alerts = readingsOutsideRange(station,
                              operatingPlan.temperatureFloor,
                              operatingPlan.temperatureCeiling,
                              null);
```

Пока что я не изменил никакого поведения, так что все тесты должны успешно проходить. Затем я перехожу к каждой точке вызова и настраиваю ее таким образом, чтобы был передан правильный диапазон.

```
Точка вызова...
const range = new NumberRange(operatingPlan.temperatureFloor,
                              operatingPlan.temperatureCeiling);
alerts = readingsOutsideRange(station,
                              operatingPlan.temperatureFloor,
                              operatingPlan.temperatureCeiling,
                              range);
```

Я все еще не изменил никакого поведения, так как параметр не используется. Все тесты должны успешно выполняться.

Теперь я могу начать заменять использование параметров. Начну с максимального значения.

```
function readingsOutsideRange(station, min, ~~max,~~ range) {
  return station.readings
    .filter(r => r.temp < min || r.temp > range.max);
}

Точка вызова...
const range = new NumberRange(operatingPlan.temperatureFloor,
                              operatingPlan.temperatureCeiling);
alerts = readingsOutsideRange(station,
                              operatingPlan.temperatureFloor,
                              ~~operatingPlan.temperatureCeiling,~~
                              range);
```

Теперь я могу выполнить тестирование и удалить второй параметр.

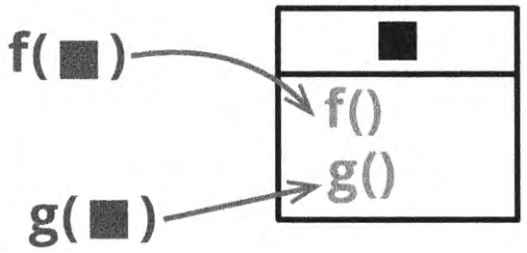
```
function readingsOutsideRange(station, ~~min,~~ range) {
  return station.readings
    .filter(r => r.temp < range.min || r.temp > range.max);
}
Точка вызова...
const range = new NumberRange(operatingPlan.temperatureFloor,
                              operatingPlan.temperatureCeiling);
alerts = readingsOutsideRange(station,
                              ~~operatingPlan.temperatureFloor,~~
                              range);
```

Рефакторинг завершен. Замена набора параметров реальным объектом — это просто настройка для действительно стоящих изменений кода. Большим преимуществом создания такого класса является то, что я могу перенести в него поведение. В нашем случае я бы добавил метод, который проверяет, попадает ли значение в данный диапазон.

```
function readingsOutsideRange(station, range) {
  return station.readings
    .filter(r => !range.contains(r.temp));
}

class NumberRange...
  contains(arg) {return (arg >= this.min && arg <= this.max);}
```

Это первый шаг к созданию диапазона [30], который может включать разнообразное полезное поведение. Определив потребность разрабатываемой программы в диапазоне, я могу продолжить поиск других случаев использования пары чисел max/min, заменяя их диапазоном. (Одной из непосредственных возможностей является рабочий план, в котором я могу заменить `TemperatureFloor` и `TemperatureCeiling` диапазоном `TemperatureRange`.) Изучая, как используются эти пары, я могу перенести более полезное поведение в класс диапазона, упрощая его использование в кодовой базе. Первое, что я могу добавить, — это метод проверки на равенство на основе значений, чтобы сделать диапазон истинным объектом-значением.

Объединение функций в класс (Combine Functions into Class)




```
function base(aReading) {...}
function taxableCharge(aReading) {...}
function calculateBaseCharge(aReading) {...}
```

```
class Reading {
  base() {...}
  taxableCharge() {...}
  calculateBaseCharge() {...}
}
```

Мотивация

В большинстве современных языков программирования классы являются фундаментальной конструкцией. Они связывают данные и функции в совместно используемую среду, предоставляя часть этих данных и функций другим элементам программы для совместной работы. Они являются основной конструкцией в объектно-ориентированных языках, но полезны и при использовании других парадигм.

Когда я вижу группу функций, которые тесно взаимодействуют с общими данными (обычно передаваемыми в качестве аргументов в вызовы функций), я вижу возможность сформировать класс. Использование класса создает общую среду, в которой совместное использование этих функций видно более отчетливо и позволяет мне упростить вызовы функций внутри объекта путем удаления множества аргументов. Кроме того, класс предоставляет возможность получения ссылки для передачи такого объекта в другие части системы.

В дополнение к организации уже сформированных функций этот рефакторинг также предоставляет хорошую возможность идентифицировать другие части вычислений и преобразовать их в методы нового класса.

Другим способом организации функций является рефакторинг *Объединение функций в преобразование* (с. 195). Какой именно из этих рефакторингов использовать, зависит от более широкого контекста программы. Одним из существенных преимуществ применения класса является то, что он позволяет клиентам изменять основные данные объекта при сохранении их согласованности.

Такие функции могут быть объединены не только в класс, но и во вложенную функцию. Обычно вложенной функции я предпочитаю класс, так как тестирование вложенных функций может оказаться достаточно сложным. Классы также необходимы, когда в группе имеется более одной функции, которую я хочу предоставить для использования другим разработчикам.

Языки, которые не имеют классов в качестве фундаментального языкового элемента, а в качестве таковых в них выступают функции, для обеспечения соответствующей функциональной возможности часто используют *функцию как объект* [24].

Техника

- Примените рефакторинг *Инкапсуляция записи* (с. 208) к общей записи данных, совместно используемой этими функциями.
- Если данные, общие для двух функций, еще не сгруппированы в структуру записи, воспользуйтесь рефакторингом *Введение объекта параметра* (с. 186) для создания записи, группирующей эти данные.

  Для каждой функции, которая использует общую запись, примените рефакторинг *Перенос функции* (с. 244), чтобы переместить ее в новый класс.
- Любые аргументы вызова функции, которые являются членами класса, могут быть удалены из списка аргументов.

  Каждый фрагмент логики, который манипулирует данными, может быть извлечен с помощью рефакторинга *Извлечение функции* (с. 152) и затем перемещен в новый класс.

Пример

Я вырос в Англии, стране, известной своей любовью к чаю. (Лично мне не нравится большинство чаев, которые подают в Англии, но с годами я приобрел вкус к китайскому и японскому чаям.) Так что моя авторская фантазия придумала программу для учета распространения чая среди населения. Каждый месяц программа считывает данные в виде записей наподобие следующей:

```
reading = {customer: "ivan", quantity: 10, month: 5, year: 2017};
```

Я просматриваю код, который обрабатывает эти записи, и вижу много мест, где с данными выполняются похожие вычисления. Поэтому я нахожу место, которое рассчитывает основную расценку:

```
Клиент 1...
const aReading = acquireReading();
const baseCharge =
  baseRate(aReading.month,aReading.year)*aReading.quantity;
```

В Англии все должно облагаться налогом, в том числе чай. Но правила позволяют освободить часть чая от налогообложения.

```
Клиент 2...
const aReading = acquireReading();
const base =
  (baseRate(aReading.month,aReading.year)*aReading.quantity);
const taxableCharge =
  Math.max(0, base - taxThreshold(aReading.year));
```

Я уверен, что вы, как и я, заметили, что формула для вычисления основной расценки в этих двух фрагментах дублируется. Надеюсь, вы уже прикидываете, как применить рефакторинг *Извлечение функции* (с. 152). Но, похоже, наша работа уже была проделана за нас в другом месте.

```
Клиент 3...
const aReading = acquireReading();
const basicChargeAmount = calculateBaseCharge(aReading);
function calculateBaseCharge(aReading) {
  return baseRate(aReading.month,aReading.year)*aReading.quantity;
}
```

С учетом этого у меня возникает естественное желание изменить два ранних фрагмента клиентского кода, чтобы использовать эту функцию. Но проблема с такими функциями верхнего уровня в том, что их часто легко не заметить. Я бы предпочел изменить код так, чтобы обеспечить более тесную связь функции с данными, которые она обрабатывает. Хороший способ сделать это — превратить данные в класс.

Чтобы превратить запись в класс, я использую рефакторинг *Инкапсуляция записи* (с. 208).

```
class Reading {
  constructor(data) {
    this._customer = data.customer;
    this._quantity = data.quantity;
    this._month = data.month;
    this._year = data.year;
  }
  get customer() {return this._customer;}
  get quantity() {return this._quantity;}
  get month() {return this._month;}
  get year() {return this._year;}
}
```

Чтобы переместить поведение, я начну с функции, которая у меня уже есть: `calculateBaseCharge`. Чтобы использовать новый класс, мне нужно применить ее к данным, как только я их получу.

```
Клиент 3...
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const basicChargeAmount = calculateBaseCharge(aReading);
```

Затем я применяю рефакторинг *Перенос функции* (с. 244) для перемещения `calculateBaseCharge` в новый класс.

```
class Reading...
get calculateBaseCharge() {
  return baseRate(this.month, this.year) * this.quantity;
}

Клиент 3...
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const basicChargeAmount = aReading.calculateBaseCharge;
```

Пока я этим занимаюсь, я использую рефакторинг *Изменение объявления функции* (с. 170), чтобы результат больше соответствовал моему вкусу.

```
get baseCharge() {
  return baseRate(this.month, this.year) * this.quantity;
}

Клиент 3...
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const basicChargeAmount = aReading.baseCharge;
```

При таком именовании клиент класса чтения не может сказать, является ли baseCharge полем или производным значением. И это хорошо, потому что соответствует принципу унифицированного доступа [38].

Теперь я изменяю код первого клиента, чтобы он вызывал метод, а не повторял расчет.

```
Клиент 1...
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const baseCharge = aReading.baseCharge;
```

Существует большая вероятность, что я буду использовать рефакторинг *Встраивание переменной* (с. 169) для переменной baseCharge еще до конца дня. Но для этого рефакторинга более важным является клиент, который рассчитывает налогооблагаемую сумму. Здесь мой первый шаг — использовать новое свойство основной расценки.

```
Клиент 2...
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const taxableCharge =
  Math.max(0, aReading.baseCharge - taxThreshold(aReading.year));
```

Я применяю рефакторинг *Извлечение функции* (с. 152) для вычисления налогооблагаемой суммы.

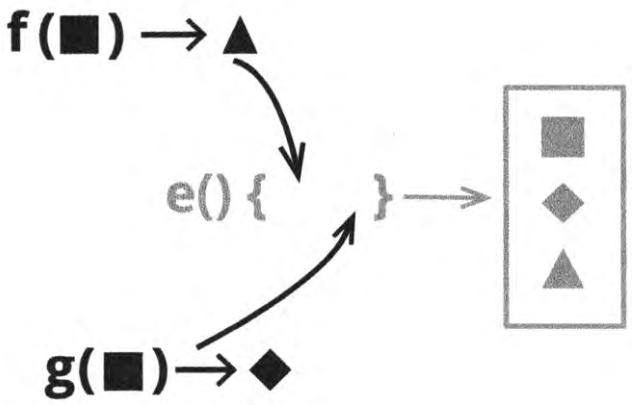
```
function taxableChargeFn(aReading) {
  return Math.max(0, aReading.baseCharge -
                     taxThreshold(aReading.year));
}

Клиент 3...
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const taxableCharge = taxableChargeFn(aReading);
```

Затем я применяю рефакторинг *Перенос функции* (с. 244).

```
class Reading...
get taxableCharge() {
  return Math.max(0, this.baseCharge -
                     taxThreshold(this.year));
}

Клиент 3...
const rawReading = acquireReading();
const aReading = new Reading(rawReading);
const taxableCharge = aReading.taxableCharge;
```

Поскольку все производные данные рассчитываются по запросу, передо мной не встает проблема, должен ли я обновлять сохраненные данные. В общем случае я предпочитаю неизменяемые данные, но имеется множество обстоятельств, которые вынуждают нас работать с изменяемыми данными (такими как языковая экосистема JavaScript, которая разрабатывалась без учета неизменяемости). Когда есть обоснованные ожидания, что где-то в программе будут обновлены данные, — класс оказывается очень полезен.

Объединение функций в преобразование (Combine Functions into Transform)




```
function base(aReading) {...}
function taxableCharge(aReading) {...}
```

```
function enrichReading(argReading) {
  const aReading = _.cloneDeep(argReading);
  aReading.baseCharge = base(aReading);
  aReading.taxableCharge = taxableCharge(aReading);
  return aReading;
}
```

Мотивация

Программное обеспечение часто включает передачу данных в программы, которые вычисляют на их основе различную производную информацию. Эти вычисляемые значения могут понадобиться в нескольких местах, так что вычисления часто повторяются везде, где используются эти производные данные.

Я предпочитаю свести все эти производные данные вместе, получив постоянное место, где их можно найти и обновить, чтобы избежать дублирования логики.

Один из способов сделать это — использовать функцию преобразования данных, которая принимает исходные данные в качестве входных и вычисляет все производные значения, помещая каждое из них в качестве поля в выходные данные. После этого, чтобы изучить полученные данные, мне нужно всего лишь рассмотреть функцию преобразования.

Альтернативой данному рефакторингу является рефакторинг *Объединение функций в класс* (с. 190), который перемещает логику в методы класса, сформированного из исходных данных. Любой из этих рефакторингов полезен, и мой выбор часто зависит от уже применяемого при разработке стиля программирования. Но есть и одно важное отличие: если исходные данные обновляются в коде, лучше использовать класс. Данный рефакторинг сохраняет производные данные в новой записи, поэтому, если исходные данные изменятся, я столкнусь с несогласованностью.

Одна из причин, по которой мне нравится данный рефакторинг, — избежание дублирования логики вычислений. Я мог бы добиться этого, просто применяя к логике рефакторинг *Извлечение функции* (с. 152), но зачастую сложно искать функции, если только они не находятся близко к структурам данных, с которыми работают. Использование преобразования (или класса) облегчает их поиск и использование.

Техника

- Создайте функцию преобразования, которая принимает преобразуемую запись и возвращает те же значения, что и исходные.

  Обычно это действие включает глубокое копирование записи. Часто стоит написать тест, чтобы убедиться, что преобразование не изменяет исходную запись.

- Выберите некоторую логику и переместите ее тело в функцию преобразования, чтобы создать новое поле в записи. Измените код клиента для обращения к новому полю.

  Если логика слишком сложная — воспользуйтесь сначала рефакторингом *Извлечение функции* (с. 152).

- Выполните тестирование.
- Повторите эти действия для других нужных функций.

Пример

Там, где я вырос, чай является важной частью жизни — настолько важной, что я могу представить себе особую утилиту, которая контролирует обеспечение населения чаем, представляющую собой своего рода отдельную коммунальную службу. Каждый месяц утилита получает сведения о том, сколько чая приобрел покупатель.

```
reading = {customer: "ivan", quantity: 10, month: 5, year: 2017};
```

Код в разных местах рассчитывает различные последствия потребления чая. Одним из таких расчетов является базовая денежная сумма, которая используется для расчета оплаты клиентом.

```
Клиент 1...
const aReading = acquireReading();
const baseCharge =
  baseRate(aReading.month, aReading.year) * aReading.quantity;
```

Другим расчетом является расчет суммы, которая должна облагаться налогом (и которая меньше базовой суммы, поскольку правительство мудро считает, что каждый гражданин должен получить некоторое количество чая беспошлинно).

```
Клиент 2...
const aReading = acquireReading();
const base =
  (baseRate(aReading.month, aReading.year) * aReading.quantity);
const taxableCharge =
  Math.max(0, base - taxThreshold(aReading.year));
```

Просматривая этот код, я вижу, что вычисления повторяются в нескольких местах. Такое дублирование вызывает проблемы, когда нужно внести изменения (и я готов спорить, что именно "когда", а не "если"). Я могу справиться с этим дублированием с помощью применения рефакторинга *Извлечение функции* (с. 152) к этим вычислениям, но такие функции часто оказываются разбросанными по программе, и это затрудняет будущим разработчикам осознание того, что они там есть. Действительно, рассматривая программу, я обнаруживаю такую функцию, используемую в совсем другой области кода.

```
Клиент 3...
const aReading = acquireReading();
const basicChargeAmount = calculateBaseCharge(aReading);

function calculateBaseCharge(aReading) {
  return baseRate(aReading.month,aReading.year)*aReading.quantity;
}
```

Один из способов решения проблемы — переместить все эти получаемые производные результаты в одну функцию преобразования, которая получает данные и вычисляет все необходимые результаты.

Я начинаю с создания функции преобразования, которая просто копирует входной объект.

```
function enrichReading(original) {
  const result = _.cloneDeep(original);
  return result;
}
```

Я использую `cloneDeep` для выполнения глубокого копирования.

Применяя преобразование, которое создает, по существу, тот же выход, что и получаемые данные, но с дополнительной информацией, — я предпочитаю использовать термин *обогащение* (enrich). Если входные данные превращаются во что-то иное — с моей точки зрения, уместнее термин *преобразование*, или *трансформация* (transform).

Затем я выбираю одно из вычислений, которое хочу изменить. Сначала я "обогащаю" входные сведения; в настоящий момент обогащение, по сути, не выполняет никаких действий.

```
Клиент 3...
const rawReading = acquireReading();
const aReading = enrichReading(rawReading);
const basicChargeAmount = calculateBaseCharge(aReading);
```

Я применяю рефакторинг *Перенос функции* (с. 244) к функции `calculateBaseCharge`, чтобы переместить ее в обогащающее вычисление.

```
function enrichReading(original) {
  const result = _.cloneDeep(original);
  result.baseCharge = calculateBaseCharge(result);
  return result;
}
```

В функции преобразования я изменяю объект результата, а не копирую его каждый раз. Мне нравится неизменяемость, но в большинстве распространенных языков программирования с ней сложно работать. Я готов приложить дополнительные усилия, чтобы поддерживать ее в границах, но допускаю изменения в более мелких областях видимости. Я также выбираю имена (используя `aReading` в качестве накапливающей переменной), упрощающие перемещение кода в функцию преобразования.

Я изменяю код клиента, который применяет эту функцию, так, чтобы он использовал обогащенное поле.

```
Клиент 3...
const rawReading = acquireReading();
```

```
const aReading = enrichReading(rawReading);
const basicChargeAmount = aReading.baseCharge;
```

Переместив все вызовы в функцию `calculateBaseCharge`, я могу вложить ее в `enrichReading`. Это сделало бы очевидным, что клиенты, которым требуется вычисленная основная расценка, должны использовать "обогащенную" запись.

Здесь следует остерегаться одной ловушки. Записывая `enrichReading` следующим образом, чтобы вернуть обогащенные сведения, я подразумеваю, что исходная запись со сведениями не изменяется. Поэтому было бы разумно добавить такой тест.

```
it('check reading unchanged', function() {
  const baseReading = {customer: "ivan", quantity: 15,
                       month: 5, year: 2017};
  const oracle = _.cloneDeep(baseReading);
  enrichReading(baseReading);
  assert.deepEqual(baseReading, oracle);
});
```

Затем я могу изменить код клиента 1 так, чтобы он использовал то же самое поле.

```
Клиент 1...
const rawReading = acquireReading();
const aReading = enrichReading(rawReading);
const baseCharge = aReading.baseCharge;
```

Вероятно, после этого я смогу использовать рефакторинг *Встраивание переменной* (с. 169) и для `baseCharge`.

Переходим к расчету налогооблагаемой суммы. Мой первый шаг состоит во внесении добавлений в функцию преобразования.

```
const rawReading = acquireReading();
const aReading = enrichReading(rawReading);
const base =
  (baseRate(aReading.month,aReading.year)*aReading.quantity);
const taxableCharge =
  Math.max(0, base - taxThreshold(aReading.year));
```

Я могу сразу же заменить расчет основной расценки новым полем. Если бы расчет был сложным, я мог бы сначала применить рефакторинг *Извлечение функции* (с. 152), но здесь все достаточно просто, чтобы сделать всю работу за один шаг.

```
const rawReading = acquireReading();
const aReading = enrichReading(rawReading);
const base = aReading.baseCharge;
const taxableCharge =
  Math.max(0, base - taxThreshold(aReading.year));
```

Протестировав выполненную работу, я перехожу к рефакторингу *Встраивание переменной* (с. 169):

```
const rawReading = acquireReading();
const aReading = enrichReading(rawReading);
const taxableCharge =
  Math.max(0, aReading.baseCharge - taxThreshold(aReading.year));
```

и перемещаю вычисления в функцию преобразования:

```
function enrichReading(original) {
  const result = _.cloneDeep(original);
  result.baseCharge = calculateBaseCharge(result);
  result.taxableCharge =
    Math.max(0, result.baseCharge - taxThreshold(result.year));
  return result;
}
```

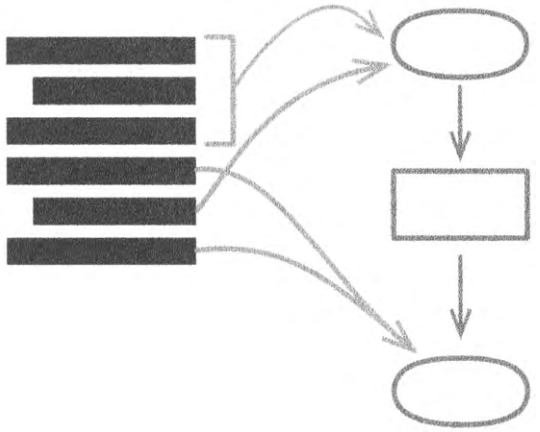
Теперь я изменяю исходный код так, чтобы он использовал новое поле.

```
const rawReading = acquireReading();
const aReading = enrichReading(rawReading);
const taxableCharge = aReading.taxableCharge;
```

После выполнения тестирования вполне вероятно, что я смогу применить рефакторинг *Встраивание переменной* (с. 169) к `taxableCharge`.

С такими обогащенными сведениями есть одна большая проблема: что произойдет, если клиент изменит значение данных? Изменение, например, значения поля количества приведет к получению противоречивых данных. Чтобы избежать этого, наилучший вариант в JavaScript — использовать рефакторинг *Объединение функций в класс* (с. 190). Если я имею дело с языком программирования с неизменяемыми структурами данных, этой проблемы у меня не возникнет, поэтому описываемый рефакторинг чаще встречаются именно в таких языках. Но даже в языках без поддержки неизменяемости я могу использовать преобразования — если данные находятся в контексте только для чтения, например при получении данных для отображения на веб-странице.

Разделение этапа (Split Phase)



```
const orderData = orderString.split(/\s+/);
const productPrice = priceList[orderData[0].split("-")[1]];
const orderPrice = parseInt(orderData[1]) * productPrice;
```

```
const orderRecord = parseOrder(order);
const orderPrice = price(orderRecord, priceList);

function parseOrder(aString) {
  const values = aString.split(/\s+/);
  return ({
    productID: values[0].split("-")[1],
    quantity: parseInt(values[1]),
  });
}
function price(order, priceList) {
  return order.quantity * priceList[order.productID];
}
```

Мотивация

Когда я сталкиваюсь с кодом, который имеет дело с двумя разными задачами, я ищу способ разделить его на отдельные модули. Я стараюсь выполнить такое разделение, поскольку, если мне нужно будет вносить изменения в программу, мне придется иметь дело с каждой задачей в отдельности и не потребуется держать их обе в голове. Если мне повезет, то, возможно, будет достаточно внести изменения только в один модуль без необходимости запоминать детали другого.

Один из лучших способов такого разделения — разделить поведение на две последовательные фазы. Хорошим примером может служить некоторая обработка данных, когда входные данные не отражают модель, необходимую для выполнения логики. Прежде чем начать основную работу, можно привести входные данные в вид, наиболее подходящий для этапа основной обработки. Или, например, вы можете взять некоторую логику в программе и разделить ее на последовательные шаги, где действия на каждом этапе существенно отличны от действий на других этапах.

Одним из наиболее очевидных примеров является компилятор. Основная его задача состоит в том, чтобы, получив некоторый текст (код на языке программирования), преобразовать его в некоторую исполняемую форму (например, объектный код для конкретной платформы). Со временем выяснилось, что эту глобальную задачу можно не без пользы для дела разбить на цепочку фаз: разделить текст на токены, превратить поток токенов в синтаксическое дерево, затем выполнить различные шаги по преобразованию синтаксического дерева (например, с целью оптимизации) и, наконец, сгенерировать объектный код. Каждый этап имеет ограниченную сферу применения, и я могу работать с одним этапом, не понимая деталей работы других.

Подобное разделение этапов распространено в больших программах. Например, различные фазы компилятора могут содержать много функций и классов. Но я могу выполнить базовый рефакторинг разделения фаз для любого фрагмента кода всякий раз, когда вижу возможность с пользой разделить код на разные фазы. Это имеет смысл, когда разные этапы фрагмента кода используют разные наборы данных и функций. Превратив их в отдельные модули, я могу сделать это различие явным.

Техника

- Выделите код второй фазы в собственную функцию.
- Выполните тестирование.
- Введите промежуточную структуру данных как дополнительный аргумент извлекаемой функции.
- Выполните тестирование.
- Изучите каждый параметр извлеченной второй фазы. Если он используется в первой фазе, переместите его в промежуточную структуру данных. Выполните тестирование после каждого переноса.

  Иногда параметр не должен использоваться на втором этапе. В этом случае извлеките результаты каждого использования параметра в поле промежуточной структуры данных и примените рефакторинг *Перенос инструкций в точку вызова* (с. 262) к строке, которая его заполняет.

- Примените рефакторинг *Извлечение функции* (с. 152) к коду первой фазы, возвращая промежуточную структуру данных.

Разумно также извлечь первую фазу в объект преобразования.

Пример

Я начну с кода для определения цены заказа на некоторые (неважно, какие) товары.

```
function priceOrder(product, quantity, shippingMethod) {
  const basePrice = product.basePrice * quantity;
  const discount = Math.max(quantity-product.discountThreshold,0)
                   *product.basePrice*product.discountRate;
  const shippingPerCase =
    (basePrice > shippingMethod.discountThreshold)
    ? shippingMethod.discountedFee : shippingMethod.feePerCase;
  const shippingCost = quantity * shippingPerCase;
  const price = basePrice - discount + shippingCost;
  return price;
}
```

Хотя это обычный тривиальный пример, здесь ощущается наличие двух фаз. Первые несколько строк кода используют информацию о товаре для расчета (ориентированной на товар) цены заказа, а более поздний код использует информацию о доставке для определения стоимости доставки. Если нужно будет внести изменения, которые усложнят расчеты цены и доставки, но при этом будут работать относительно независимо, — разделение этого кода на две фазы окажется полезным.

Я начинаю с применения рефакторинга *Извлечение функции* (с. 152) к вычислениям, связанным с доставкой.

```
function priceOrder(product, quantity, shippingMethod) {
  const basePrice = product.basePrice * quantity;
  const discount = Math.max(quantity-product.discountThreshold,0)
                   *product.basePrice*product.discountRate;
  const price = applyShipping(basePrice, shippingMethod,
                              quantity, discount);
  return price;
}
function applyShipping(basePrice, shippingMethod,
                       quantity, discount) {
  const shippingPerCase =
    (basePrice > shippingMethod.discountThreshold)
    ? shippingMethod.discountedFee : shippingMethod.feePerCase;
  const shippingCost = quantity * shippingPerCase;
  const price = basePrice - discount + shippingCost;
  return price;
}
```

Я передаю все данные, необходимые для второй фазы, в качестве отдельных параметров. В более реалистичном случае их может быть очень много, но я не беспокоюсь об этом, так как позже все они будут опущены.

Далее я ввожу промежуточную структуру данных, которая будет обеспечивать взаимодействие между этими двумя фазами.

```
function priceOrder(product, quantity, shippingMethod) {
  const basePrice = product.basePrice * quantity;
  const discount = Math.max(quantity-product.discountThreshold,0)
                   * product.basePrice * product.discountRate;
  const priceData = {};
  const price = applyShipping(priceData, basePrice,
                              shippingMethod, quantity, discount);
  return price;
}
function applyShipping(priceData, basePrice, shippingMethod,
                       quantity, discount) {
  const shippingPerCase =
    (basePrice > shippingMethod.discountThreshold)
    ? shippingMethod.discountedFee : shippingMethod.feePerCase;
  const shippingCost = quantity * shippingPerCase;
  const price = basePrice - discount + shippingCost;
  return price;
}
```

Теперь я рассматриваю различные параметры функции `applyShipping`. Первый из них — `basePrice`, который создается кодом первой фазы. Поэтому я переношу его в промежуточную структуру данных и удаляю его из списка параметров.

```
function priceOrder(product, quantity, shippingMethod) {
  const basePrice = product.basePrice * quantity;
  const discount = Math.max(quantity-product.discountThreshold,0)
                   * product.basePrice * product.discountRate;
  const priceData = {basePrice: basePrice};
  const price = applyShipping(priceData, ~~basePrice,~~
                              shippingMethod, quantity, discount);
  return price;
}
function applyShipping(priceData, ~~basePrice,~~ shippingMethod,
                       quantity, discount) {
  const shippingPerCase =
    (priceData.basePrice > shippingMethod.discountThreshold)
    ? shippingMethod.discountedFee : shippingMethod.feePerCase;
  const shippingCost = quantity * shippingPerCase;
  const price = priceData.basePrice - discount + shippingCost;
  return price;
}
```

Следующий параметр в списке — shippingMethod. Этот параметр я оставляю как есть, так как он не используется кодом первой фазы.

Следующим идет параметр quantity. Он используется на первой фазе, но не создается ею, поэтому я мог бы оставить его в списке параметров. Однако я предпочитаю переместить в промежуточную структуру данных как можно больше параметров.

```
function priceOrder(product, quantity, shippingMethod) {
  const basePrice = product.basePrice * quantity;
  const discount = Math.max(quantity-product.discountThreshold,0)
                   * product.basePrice * product.discountRate;
  const priceData = {basePrice: basePrice, quantity: quantity};
  const price = applyShipping(priceData, shippingMethod,
                              ~~quantity,~~ discount);
  return price;
}
function applyShipping(priceData, shippingMethod,
                       ~~quantity,~~ discount) {
  const shippingPerCase =
    (priceData.basePrice > shippingMethod.discountThreshold)
    ? shippingMethod.discountedFee : shippingMethod.feePerCase;
  const shippingCost = priceData.quantity * shippingPerCase;
  const price = priceData.basePrice - discount + shippingCost;
  return price;
}
```

Далее я делаю то же самое с параметром discount.

```
function priceOrder(product, quantity, shippingMethod) {
  const basePrice = product.basePrice * quantity;
  const discount = Math.max(quantity-product.discountThreshold,0)
                   * product.basePrice * product.discountRate;
const priceData = {basePrice: basePrice, quantity: quantity,
                   discount:discount};
const price = applyShipping(priceData, shippingMethod~~, discount~~);
return price;
}
function applyShipping(priceData, shippingMethod~~, discount~~) {
  const shippingPerCase =
    (priceData.basePrice > shippingMethod.discountThreshold)
    ? shippingMethod.discountedFee : shippingMethod.feePerCase;
  const shippingCost = priceData.quantity * shippingPerCase;
  const price = priceData.basePrice - priceData.discount
              + shippingCost;
  return price;
}
```

После прохода по всем параметрам функции у меня полностью сформирована промежуточная структура данных. Так что теперь я могу извлечь код первой фазы в отдельную функцию, возвращающую эти данные.

```
function priceOrder(product, quantity, shippingMethod) {
  const priceData = calculatePricingData(product, quantity);
  const price = applyShipping(priceData, shippingMethod);
  return price;
}
function calculatePricingData(product, quantity) {
  const basePrice = product.basePrice * quantity;
  const discount = Math.max(quantity-product.discountThreshold,0)
                   * product.basePrice * product.discountRate;
  return {basePrice: basePrice, quantity: quantity,
          discount:discount};
}
function applyShipping(priceData, shippingMethod) {
  const shippingPerCase =
    (priceData.basePrice > shippingMethod.discountThreshold)
    ? shippingMethod.discountedFee : shippingMethod.feePerCase;
  const shippingCost = priceData.quantity * shippingPerCase;
  const price = priceData.basePrice - priceData.discount
                + shippingCost;
  return price;
}
```

Я не могу сопротивляться искушению нанести на картину последние штрихи.

```
function priceOrder(product, quantity, shippingMethod) {
  const priceData = calculatePricingData(product, quantity);
  return applyShipping(priceData, shippingMethod);
}
function calculatePricingData(product, quantity) {
  const basePrice = product.basePrice * quantity;
  const discount = Math.max(quantity-product.discountThreshold,0)
                   * product.basePrice * product.discountRate;
  return {basePrice: basePrice, quantity: quantity,
          discount:discount};
}
function applyShipping(priceData, shippingMethod) {
  const shippingPerCase =
    (priceData.basePrice > shippingMethod.discountThreshold)
    ? shippingMethod.discountedFee : shippingMethod.feePerCase;
  const shippingCost = priceData.quantity * shippingPerCase;
  return priceData.basePrice - priceData.discount + shippingCost;
}
```

Глава 7

Инкапсуляция

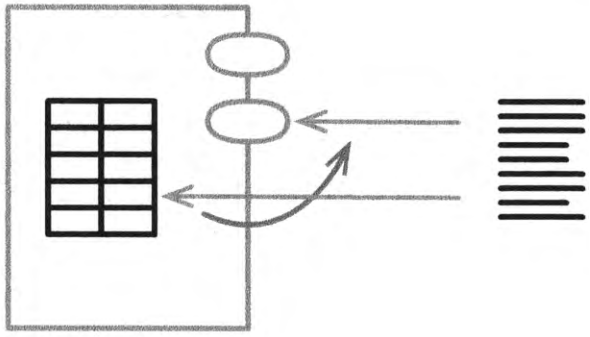
Возможно, наиболее важный критерий, который следует использовать при декомпозиции модулей, — это определение "секретов", которые модули должны скрывать от остальной части системы [45]. Наиболее распространенными секретами являются структуры данных, которые можно скрыть, инкапсулируя их с помощью рефакторингов *Инкапсуляция записи* (с. 208) и *Инкапсуляция коллекции* (с. 216). Даже примитивные данные-значения могут быть инкапсулированы с помощью рефакторинга *Замена примитива объектом* (с. 221), величина выгоды от которого часто удивляет разработчиков. Временные переменные часто мешают рефакторингу: я должен убедиться, что они вычисляются в правильном порядке, а их значения доступны для других частей кода, которые в них нуждаются. Здесь может помочь рефакторинг *Замена временной переменной запросом* (с. 225), в особенности при разделении слишком длинной функции.

Классы предназначены для сокрытия информации. В предыдущей главе я описал способ их формирования с помощью рефакторинга *Объединение функций в класс* (с. 190). Распространенные операции извлечения/встраивания функций применимы и к классам — для этого имеются рефакторинги *Извлечение класса* (с. 229) и *Встраивание класса* (с. 232).

Помимо сокрытия внутренних элементов классов, часто полезно скрывать связи между классами. Помочь в этом может рефакторинг *Сокрытие делегата* (с. 235). Однако слишком большое количество сокрытий приводит к раздутым интерфейсам, поэтому на вооружении стоит иметь и обратный рефакторинг — *Удаление посредника* (с. 238).

Классы и модули являются наибольшими формами инкапсуляции, но функции также выполняют инкапсуляцию своих реализаций. Иногда может понадобиться выполнить большое количество изменений в алгоритме; это можно сделать, обернув его в функцию с помощью рефакторинга *Извлечение функции* (с. 152) и применив рефакторинг *Подстановка алгоритма* (с. 240).

Инкапсуляция записи (Encapsulate Record)



Бывший рефакторинг *Замена записи классом данных*

```
organization = {name: "Acme Gooseberries", country: "GB"};
```

```
class Organization {
  constructor(data) {
    this._name = data.name;
    this._country = data.country;
  }
  get name() {return this._name;}
  set name(arg) {this._name = arg;}
  get country() {return this._country;}
  set country(arg) {this._country = arg;}
}
```

Мотивация

Почему для изменяемых данных я часто отдаю предпочтение объектам, а не записям? Потому что с помощью объектов я могу скрыть то, что сохраняю, и предоставить методы для всех значений. Пользователю объекта не нужно знать или заботиться о том, что в нем хранится; достаточно знать, что рассчитывается. Инкапсуляция помогает и с переименованием: я могу переименовывать поле, предоставляя методы как для новых, так и для старых имен, и постепенно обновляя все вызовы методов.

Я только что сказал, что предпочитаю использовать для *изменяемых* данных объекты. Если у меня есть неизменяемое значение, я могу просто указать все значения в единой записи, используя при необходимости шаг обогащения. Точно так же при переименовании легко сделать копию поля.

У меня может быть два вида структур записей: те, в которых я объявляю конкретные корректные имена полей, и те, которые позволяют мне использовать

произвольные имена полей. Последние часто реализуются через библиотечный класс, носящий название наподобие хеша, отображения, хеш-таблицы, словаря или ассоциативного массива. Многие языки программирования предоставляют удобный синтаксис для таких отображений, что делает их полезными во многих ситуациях. Недостатком их использования является то, что они не имеют ясного представления о своих полях. Единственный способ определить, использует ли их реализация пару начало-конец или пару начало-длина, — это посмотреть код, где они создаются и используются. Это не проблема, если используются они только в небольшом разделе программы, но чем шире область их применения, тем бо́льшую проблему я получаю от неявности их структуры. Я мог бы преобразовать такие неявные записи в явные, но, если ситуация такова, что мне нужно к этому прибегнуть, — я бы вместо такого преобразования сделал их классами.

Зачастую информация передается в виде вложенных структуры списков и хеш-отображений, которые обычно сериализуются в такие форматы, как JSON или XML. Такие структуры также могут быть инкапсулированы, что помогает в ситуации, если их форматы позже будут изменены, или если я беспокоюсь о трудно отслеживаемых обновлениях данных.

Техника

- Примените рефакторинг *Инкапсуляция переменной* (с. 178) к переменной, в которой хранится запись.

  Давайте инкапсулирующим записи функциям имена, которые легко найти.

- Замените содержимое переменной простым классом-оболочкой для записи. Определите метод доступа внутри этого класса, который возвращает необработанную запись. Внесите в функции, которые инкапсулируют переменную, изменения таким образом, чтобы они использовали этот метод доступа.
- Выполните тестирование.
- Предоставьте новые функции, которые возвращают объект, а не необработанную запись.
- Для каждого использования записи замените функцию, которая возвращает запись, функцией, которая возвращает объект. Используйте методы доступа к объекту, чтобы получать данные из полей; при необходимости — создавайте такие методы. Выполните тестирование после каждого изменения.

  В случае сложной записи, например с вложенной структурой, сосредоточьтесь сначала на клиентах, которые обновляют данные. Для клиентов, которые только читают данные, рассмотрите возможность возврата копии данных или прокси-объекта только для чтения.

- Удалите метод доступа к необработанным данным класса и легко разыскиваемые функции, которые возвращают необработанную запись.
- Выполните тестирование.
- Если поля записи сами являются структурами, подумайте о рекурсивном применении рефакторингов *Инкапсуляция записи* и *Инкапсуляция коллекции* (с. 216).

Пример

Я начну с константы, которая широко используется в программе.

```
const organization = {name: "Acme Gooseberries", country: "GB"};
```

Это объект JavaScript, который используется в качестве структуры записи различными частями программы, с обращениями к нему наподобие следующего:

```
result += `<h1>${organization.name}</h1>`;
```

и

```
organization.name = newName;
```

Первый шаг состоит в простом рефакторинге *Инкапсуляция переменной* (с. 178).

```
function getRawDataOfOrganization() {return organization;}

Пример чтения...
result += `<h1>${getRawDataOfOrganization().name}</h1>`;

Пример записи...
getRawDataOfOrganization().name = newName;
```

Это не совсем стандартный рефакторинг *Инкапсуляция переменной* (с. 178), так как я дал методу получения значения имя, специально выбранное таким образом, чтобы оно было достаточно безобразным, и его было легко найти. Это потому, что я планирую для него очень короткую жизнь.

Инкапсуляция записи означает, что требуется не просто ее сокрытие — нужен контроль над работой с ней. Я могу получить этот контроль, заменив запись классом.

```
class Organization...
class Organization {
  constructor(data) {
    this._data = data;
  }
}
```

```
Верхний уровень...
const organization =
  new Organization({name: "Acme Gooseberries", country: "GB"});

function getRawDataOfOrganization() {return organization._data;}
function getOrganization() {return organization;}
```

Теперь, когда у меня есть объект, я начинаю поиск использований записи. Любое обновление записи заменяется методом установки.

```
class Organization...
set name(aString) {this._data.name = aString;}

Клиент...
getOrganization().name = newName;
```

Точно так же я заменяю все обращения к записи для чтения соответствующим методом доступа.

```
class Organization...
get name() {return this._data.name;}

Клиент...
result += `<h1>${getOrganization().name}</h1>`;
```

Сделав это, я смогу могу выполнить обещание короткой жизни функции с уродливым именем.

```
~~function getRawDataOfOrganization() {return organization._data;}~~
function getOrganization() {return organization;}
```

Я также склонен внедрить поле `_data` непосредственно в объект.

```
class Organization {
  constructor(data) {
    this._name = data.name;
    this._country = data.country;
  }
  get name() {return this._name;}
  set name(aString) {this._name = aString;}
  get country() {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}
```

Преимущество этого решения — в разрыве связи с записью входных данных. Оно может быть полезно при наличии ссылок, которые могут нарушить инкапсуляцию. Если данные не размещены в отдельных полях, разумно выполнять копирование `_data` при присваивании.

Пример: инкапсуляция вложенной записи

В приведенном выше примере рассматривается поверхностная запись. Но что делать с глубоко вложенными данными, например получаемыми из документа формата JSON? Основные этапы рефакторинга остаются применимыми, и я точно так же должен быть осторожен с обновлениями; однако при чтении у меня появляются некоторые варианты.

В качестве примера рассмотрим немного более глубоко вложенные данные: набор клиентов, хранимый в хеш-отображении, индексируемом идентификаторами клиентов.

```
"1920": {
  name: "martin",
  id: "1920",
  usages: {
    "2016": {
      "1": 50,
      "2": 55,
      // Остальные месяцы года
    },
    "2015": {
      "1": 70,
      "2": 63,
      // Остальные месяцы года
    }
  }
},
"38673": {
  name: "neal",
  id: "38673",
  // Другие аналогичные записи клиентов
```

При таких более глубоко вложенных данных процедуры чтения и записи могут глубоко закапываться в структуру данных.

```
Пример обновления...
customerData[customerID].usages[year][month] = amount;

Пример чтения...
function compareUsage (customerID, laterYear, month) {
  const later =
        customerData[customerID].usages[laterYear][month];
  const earlier =
        customerData[customerID].usages[laterYear - 1][month];
  return {laterAmount: later, change: later - earlier};
}
```

Для инкапсуляции этих данных я начинаю работу с выполнения рефакторинга *Инкапсуляция переменной* (с. 178).

```
function getRawDataOfCustomers() {return customerData;}
function setRawDataOfCustomers(arg) {customerData = arg;}

Пример обновления...
getRawDataOfCustomers()[customerID].usages[year][month] = amount;

Пример чтения...
function compareUsage (customerID, laterYear, month) {
  const later =
    getRawDataOfCustomers()[customerID].usages[laterYear][month];
  const earlier =
    getRawDataOfCustomers()[customerID].usages[laterYear-1][month];
  return {laterAmount: later, change: later - earlier};
}
```

Затем я создаю класс для структуры данных в целом.

```
class CustomerData {
  constructor(data) {
    this._data = data;
  }
}

Верхний уровень...
function getCustomerData() {return customerData;}
function getRawDataOfCustomers() {return customerData._data;}
function setRawDataOfCustomers(arg) {
    customerData = new CustomerData(arg);}
```

Наиболее важной областью, с которой приходится работать, являются обновления. Так что, когда я ищу вызовы `getRawDataOfCustomers`, я в первую очередь рассматриваю те из них, где изменяются данные. Напомню вам, как выглядит такое обновление:

```
Пример обновления...
getRawDataOfCustomers()[customerID].usages[year][month] = amount;
```

Техника гласит, что вы должны вернуть полного клиента и использовать метод доступа, при необходимости создавая его. У меня нет методов установки для этого обновления, так что коду приходится глубоко "закапываться" в структуру. Чтобы создать нужный мне метод, я начинаю с рефакторинга *Извлечение функции* (с. 152) для кода, который "копается" в структуре данных.

```
Пример обновления...
setUsage(customerID, year, month, amount);

Верхний уровень...
function setUsage(customerID, year, month, amount) {
  getRawDataOfCustomers()[customerID].usages[year][month] = amount;
}
```

Затем я применяю рефакторинг *Перенос функции* (с. 244) для перемещения его в новый класс данных клиента.

```
Пример обновления...
getCustomerData().setUsage(customerID, year, month, amount);

class CustomerData...
setUsage(customerID, year, month, amount) {
  this._data[customerID].usages[year][month] = amount;
}
```

При работе с большой структурой данных я предпочитаю сосредоточиваться на обновлениях. Сделать их видимыми и собрать в одном месте — вот наиболее важная часть инкапсуляции.

В какой-то момент я могу решить, что собрал их все, но как я могу быть в этом уверен? Есть несколько способов это проверить. Одним из них является изменение функции `getRawDataOfCustomers` так, чтобы она возвращала глубокую копию данных; если охват программы тестами достаточно полный — то, если я пропустил какое-то изменение данных, тестирование должно это выявить.

```
Верхний уровень...
function getCustomerData() {return customerData;}
function getRawDataOfCustomers() {return customerData.rawData;}
function setRawDataOfCustomers(arg) {
    customerData = new CustomerData(arg);}

class CustomerData...
get rawData() {
  return _.cloneDeep(this._data);
}
```

Для создания глубокой копии я использую библиотеку lodash.

Другой подход состоит в том, чтобы для структуры данных вернуть доступный только для чтения прокси-объект. Такой прокси-объект может сгенерировать исключение, если клиентский код попытается изменить базовый объект. В некоторых языках это легко сделать, но в JavaScript это достаточно трудно и неприятно, так что я оставлю это в качестве упражнения для читателя. Можно также сохранить копию и рекурсивно ее "заморозить", чтобы затем обнаружить любые изменения.

Работа с обновлениями, конечно, важна, но что делать с чтением данных? Здесь есть несколько вариантов.

Вариант первый — поступить так же, как с функциями установки значений. Извлеките все операции чтения в их собственные функции и переместите их в класс данных клиента.

```
class CustomerData...
usage(customerID, year, month) {
```

```
  return this._data[customerID].usages[year][month];
}

Верхний уровень...
function compareUsage (customerID, laterYear, month) {
  const later =
    getCustomerData().usage(customerID, laterYear, month);
  const earlier =
    getCustomerData().usage(customerID, laterYear - 1, month);
  return {laterAmount: later, change: later - earlier};
}
```

Отличительной особенностью этого подхода является то, что он предоставляет для класса `customerData` явный API-интерфейс, который фиксирует все применения данного класса. Я могу посмотреть на класс и увидеть все использования его данных. Однако для многих частных случаев это может потребовать большое количество кода. Современные языки позволяют проводить в структуре данных поиск *list-and-hash* [26], поэтому имеет смысл предоставить клиентам для работы именно такую структуру данных.

Если клиенту нужна структура данных, я могу просто предоставить ему фактические данные. Проблема, однако, в том, что нет никакого способа предотвратить непосредственное изменение данных клиентами, что нарушает весь смысл инкапсуляции обновлений внутри функций. Следовательно, самое простое, что можно сделать, — это предоставлять копию базовых данных, используя написанный мною ранее метод `rawData`.

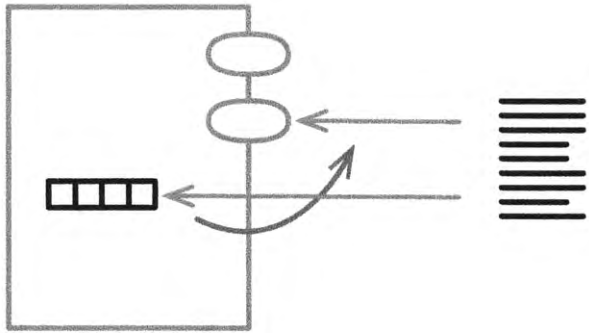
```
class CustomerData...
get rawData() {
  return _.cloneDeep(this._data);
}

Верхний уровень...
function compareUsage (customerID, laterYear, month) {
  const later =
    getCustomerData().rawData[customerID]
                     .usages[laterYear][month];
  const earlier =
    getCustomerData().rawData[customerID]
                     .usages[laterYear-1][month];
  return {laterAmount: later, change: later - earlier};
}
```

Хотя этот способ прост, он не лишен недостатков. Наиболее очевидной проблемой является стоимость копирования большой структуры данных, которая может привести к проблемам производительности. Однако затраты на производительность могут быть и приемлемыми, как минимум их влияние следует тщательно измерить, прежде чем начинать беспокоиться. Может также возникнуть

путаница, если клиенты ожидают, что изменение копии данных приведет к изменению оригинала. В случае изменений доступный только для чтения прокси-объект или замораживание скопированных данных может привести к (полезной) ошибке.

Другой вариант требует большего количества работы, но при этом предлагает наибольшую степень контроля. Он заключается в рекурсивном применении рассматриваемого рефакторинга *Инкапсуляция записи*. При этом я превращаю запись клиента в ее собственный класс, применяю рефакторинг *Инкапсуляция коллекции* (с. 216) к ее использованиям и создаю соответствующий используемый класс. После этого я могу обеспечить контроль над обновлениями с помощью методов доступа, возможно, с применением рефакторинга *Замена ссылки значением* (с. 296) к объектам использования. Но это может потребовать больших усилий и большой структуры данных — что фактически излишне, если обращений к большей части структуры данных нет. Иногда может сработать разумное сочетание методов получения значений и новых классов с использованием методов получения значений для углубления в структуру, но с возвратом объекта, который представляет собой оболочку для структуры, а не неинкапсулированных данных. Я писал об этом в статье *Рефакторинг кода загрузки документа* [31].

Инкапсуляция коллекции (Encapsulate Collection)



```
class Person {
  get courses() {return this._courses;}
  set courses(aList) {this._courses = aList;}
```

```
class Person {
  get courses() {return this._courses.slice();}
  addCourse(aCourse) { ... }
  removeCourse(aCourse) { ... }
```

Мотивация

В своих программах я предпочитаю инкапсулировать любые изменяемые данные. Это облегчает понимание того, когда и как изменяются структуры данных, а также облегчает изменение этих структур данных при необходимости. Инкапсуляция — обычная технология, в особенности в объектно-ориентированном программировании, но при работе с коллекциями часто возникают ошибки. Доступ к переменной коллекции может быть инкапсулирован, но если метод доступа возвращает саму коллекцию, то члены этой коллекции могут быть изменены без ведома класса.

Чтобы избежать этого, я предоставляю методы модификации коллекции — обычно для добавления и удаления элементов — в самом классе. Таким образом, изменения в коллекции выполняются через класс-владелец, что дает мне возможность выполнять такие изменения по мере развития программы.

Если в команде разработчиков имеется привычка не изменять коллекции вне исходного модуля, предоставления таких методов должно быть достаточно. Однако полагаться на такие привычки, как правило, неразумно; результатом могут оказаться трудно отслеживаемые ошибки. Наилучшим подходом является гарантия того, что метод доступа не возвращает просто необработанную коллекцию, так что клиенты просто не смогут случайно ее изменить.

Один из способов предотвратить изменение базовой коллекции — никогда не возвращать значение коллекции. При таком подходе любое использование поля коллекции выполняется с помощью специальных методов класса-владельца, меняющих `aCustomer.orders.size` на `aCustomer.numberOfOrders`. Я не согласен с таким подходом. Современные языки имеют богатые классы коллекций со стандартизованными интерфейсами, которые можно комбинировать разными способами, например как конвейеры коллекций [19]. Использование специальных методов для обработки такого рода функциональности добавляет большое количество дополнительного кода и усложняет применимость комбинирования операций над коллекциями.

Другой способ состоит в разрешении доступа к коллекции только для чтения. Java, например, позволяет легко вернуть прокси-объект для коллекции, доступный только для чтения. Такой прокси-объект перенаправляет все операции чтения к базовой коллекции и блокирует все попытки записи (в случае Java генерируя исключения). Подобный подход используется библиотеками, которые основывают композиции коллекций на некоторой разновидности итераторов или перечислений (при условии, что такой итератор не может модифицировать базовую коллекцию).

Вероятно, наиболее распространенный подход состоит в предоставлении метода получения коллекции, возвращающего копию базовой коллекции. Очевидно,

что любые изменения копии не влияют на инкапсулированную коллекцию. Это может вызвать определенную путаницу, если программисты преднамеренно вносят изменения и ожидают, что при этом будут изменены исходные данные, но во многих кодовых базах используются методы получения коллекций, возвращающие копии. Если коллекция имеет большой размер, это может оказаться проблемой с точки зрения производительности и используемых ресурсов, но в большинстве случаев списки не так уж велики, поэтому в общем случае следует применять обычные правила для производительности (см. раздел "Рефакторинг и производительность" главы 2, "Принципы рефакторинга").

Еще одно различие между использованием прокси-объекта и копии заключается в том, что модификация исходных данных будет видимой в случае прокси-объекта, но не в случае использования копии. В большинстве ситуаций это не составляет проблемы, поскольку списки, к которым обращаются таким образом, обычно хранятся очень непродолжительное время.

Самое важное здесь — согласованность в кодовой базе. Используйте только один механизм, чтобы каждый разработчик мог привыкнуть к его поведению и ожидать это поведение при вызове любой функции доступа к коллекции.

Техника

- Если обращение к коллекции еще не инкапсулировано — примените рефакторинг *Инкапсуляция переменной* (с. 178).
- Добавьте функции для вставки элементов в коллекцию и удаления их из нее.

  Если для коллекции имеется функция изменения значения, по возможности используйте рефакторинг *Удаление метода установки значения* (с. 376). Если же таковой нет, сделайте ее принимающей копию предоставленной коллекции.
- Выполните статические проверки.
- Найдите все обращения к коллекции. Если где-то вызываются функции, модифицирующие коллекцию, — измените вызовы таким образом, чтобы они использовали новые функции добавления/удаления элементов коллекции. Проводите тестирование после каждого изменения.
- Измените метод получения значения для коллекции таким образом, чтобы он возвращал защищенное ее представление, используя прокси-объект только для чтения или копию коллекции.
- Выполните тестирование.

Пример

Я начину с класса человека (`Person`), в котором есть поле списка курсов.

```
class Person...
  constructor (name) {
    this._name = name;
    this._courses = [];
  }
  get name() {return this._name;}
  get courses() {return this._courses;}
  set courses(aList) {this._courses = aList;}

class Course...
  constructor(name, isAdvanced) {
    this._name = name;
    this._isAdvanced = isAdvanced;
  }
  get name() {return this._name;}
  get isAdvanced() {return this._isAdvanced;}
```

Клиенты используют коллекцию курсов для сбора информации о курсах.

```
numAdvancedCourses = aPerson.courses
    .filter(c => c.isAdvanced)
    .length
;
```

Простодушный разработчик может сказать, что этот класс имеет надлежащую инкапсуляцию данных: в конце концов, каждое поле защищено методами доступа. Но я готов спорить и утверждаю, что список курсов должным образом не инкапсулирован — ведь любой, кто обновляет курсы как единое значение, получает реальный контроль через метод установки значения:

```
Код клиента...
const basicCourseNames = readBasicCourseNames(filename);
aPerson.courses = basicCourseNames.map(name => new Course(name,
false));
```

Но клиентам может оказаться проще обновить список курсов непосредственно.

```
Код клиента...
for(const name of readBasicCourseNames(filename)) {
  aPerson.courses.push(new Course(name, false));
}
```

Это нарушает принцип инкапсуляции, потому что класс `Person` не имеет возможности контролировать обновление списка таким образом. Несмотря на то что ссылка на поле инкапсулирована, к содержимому поля это не относится.

Я начну создавать правильную инкапсуляцию, добавляя в класс `Person` методы, позволяющие клиенту добавлять и удалять отдельные курсы.

```
class Person...
addCourse(aCourse) {
  this._courses.push(aCourse);
}
removeCourse(aCourse,fnIfAbsent=() => {throw new RangeError();}){
  const index = this._courses.indexOf(aCourse);
  if (index === -1) fnIfAbsent();
  else this._courses.splice(index, 1);
}
```

Что касается удаления, я должен решить, что делать, если клиент запрашивает удаление элемента, которого нет в коллекции. Я могу либо просто пожать плечами, либо сгенерировать ошибку. Приведенный код по умолчанию генерирует исключение, но вызывающим его клиентам предоставляется возможность сделать что-либо иное, если они того пожелают.

Затем я изменяю любой код, который непосредственно модифицирует коллекцию, так, чтобы он использовал новые методы.

```
Код клиента...
for(const name of readBasicCourseNames(filename)) {
  aPerson.addCourse(new Course(name, false));
}
```

При использовании отдельных методов добавления и удаления элементов обычно нет необходимости в методе `setCourses`, так что я применяю к нему рефакторинг *Удаление метода установки значения* (с. 376). Если API по каким-либо причинам требует метод установки значения, я должен убедиться, что он помещает в поле копию коллекции.

```
class Person...
set courses(aList) {this._courses = aList.slice();}
```

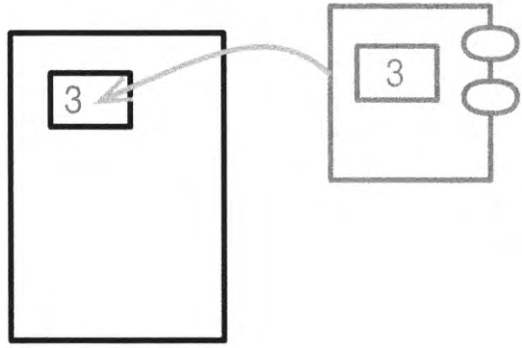
Все это позволяет клиентам использовать правильные методы модификации. Однако я предпочитаю убедиться, что никаких изменений списка без их применения не выполняется. Я могу сделать это, предоставляя копию:

```
class Person...
get courses() {return this._courses.slice();}
```

В общем случае я считаю разумным быть немного параноиком по отношению к коллекциям и предпочитаю излишнее их копирование поискам и отладкам ошибок из-за неожиданных изменений коллекций. Изменения коллекций не всегда очевидны; например, сортировка массива в JavaScript изменяет оригинал, в то время как многие языки программирования для операции, которая изменяет коллекцию, по умолчанию делают копию. Любой класс, который отвечает за управление коллекцией, всегда должен выдавать копии, однако я имею привычку выполнять копирование всегда, когда делаю что-то, что может изменить коллекцию.

Замена примитива объектом (Replace Primitive with Object)

Бывший рефакторинг *Замена значения данных объектом*
Бывший рефакторинг *Замена кода типа классом*




```
orders.filter(o => "high" === o.priority
                || "rush" === o.priority);
```

```
orders.filter(o => o.priority.higherThan(new Priority("normal")))
```

Мотивация

Часто на ранних стадиях разработки принимается решение о представлении простых фактов в виде простых элементов данных, таких как числа или строки. По мере развития программы простые вещи перестают быть столь простыми — так, некоторое время телефонный номер может быть представлен в виде строки, но позже ему потребуется специальное поведение для форматирования, извлечения кода города и тому подобного. Такая логика может быстро закончиться дублированием в кодовой базе.

Как только я понимаю, что хочу заняться чем-то отличным от простого вывода на экран, я предпочитаю создавать новый класс для соответствующих данных. Поначалу такой класс не делает ничего большего, чем служит простой примитивной оберткой для данных, но как только у меня появляется такой класс, — появляется место для поведения, соответствующее их нуждам. Так маленькие значения могут превратиться в полезные инструменты. Они могут выглядеть достаточно скромно, но их влияние на кодовую базу может оказаться на удивление большим. Многие опытные разработчики считают, что это один из наиболее ценных рефакторингов, даже если он кажется нелогичным программисту-новичку.

Техника

- Примените рефакторинг *Инкапсуляция переменной* (с. 178), если он еще не был применен.
- Создайте простой класс-значение для данных. Он должен принимать существующее значение в конструкторе и предоставлять для этого значения метод доступа.
- Выполните статические проверки.
- Измените метод установки значения таким образом, чтобы он создавал новый экземпляр класса-значения и сохранял его в поле, изменяя тип поля, если таковое имеется.
- Измените метод получения значения таким образом, чтобы он возвращал результат вызова метода получения значения нового класса.
- Выполните тестирование.
- Рассмотрите применение рефакторинга *Изменение объявления функции* (с. 170) к исходному методу доступа для лучшего отражения его действий.
- Рассмотрите возможность уточнения роли нового объекта в качестве значения или ссылочного объекта, применяя рефакторинг *Замена ссылки значением* (с. 296) или *Замена значения ссылкой* (с. 300).

Пример

Я начну с простого класса заказа, который считывает свои данные из записи с простой структурой. Одним из его свойств является приоритет, который считывается как простая строка.

```
class Order...
constructor(data) {
  this.priority = data.priority;
  // Прочие инициализации
```

Клиентский код использует его следующим образом:

```
Клиент...
highPriorityCount = orders.filter(o => "high" === o.priority
                                    || "rush" === o.priority)
                          .length;
```

Всякий раз, когда я работаю со значением данных, первое, что я делаю — это применяю к нему рефакторинг *Инкапсуляция переменной* (с. 178).

```
class Order...
get priority() {return this._priority;}
set priority(aString) {this._priority = aString;}
```

Строка конструктора, инициализирующая значение приоритета, теперь использует определенный выше метод установки значения.

Таким образом выполняется самоинкапсуляция поля, так что я могу сохранить его текущее использование во время работы с самими данными.

Я создаю простой класс-значение для приоритета. У него есть конструктор и функция преобразования, возвращающая строку.

```
class Priority {
  constructor(value) {this._value = value;}
  toString() {return this._value;}
}
```

Я предпочитаю использовать функцию преобразования (`toString`), а не метод получения значения (`value`). Для клиентов класса запрос строкового представления должен походить на преобразование, а не на получение значения свойства.

Затем, чтобы использовать новый класс, я изменяю методы доступа.

```
class Order...
  get priority() {return this._priority.toString();}
  set priority(aString) {this._priority = new Priority(aString);}
```

Теперь, когда у меня есть класс приоритета, я обнаруживаю, что текущий метод доступа в заказе вводит в заблуждение. Он возвращает не приоритет, а строку, описывающую приоритет. Поэтому мой следующий шаг состоит в использовании рефакторинга *Изменение объявления функции* (с. 170).

```
class Order...
  get priorityString() {return this._priority.toString();}
  set priority(aString) {this._priority = new Priority(aString);}

Клиент...
highPriorityCount = orders.filter(o => "high" === o.priorityString
                                    || "rush" === o.priorityString)
                          .length;
```

В этом случае я рад сохранить имя метода установки значения. Название аргумента сообщает, какое именно значение ожидается этим методом.

На этом я формально заканчиваю рефакторинг. Но, рассматривая использование приоритета, я думаю — а не следует ли использовать здесь класс приоритета? В результате я предоставляю для заказа метод доступа, который выдает непосредственно новый объект приоритета.

```
class Order...
  get priority() {return this._priority;}
  get priorityString() {return this._priority.toString();}
  set priority(aString) {this._priority = new Priority(aString);}
```

```
Клиент...
highPriorityCount = orders.filter(o =>
                                  "high" === o.priority.toString()
                               || "rush" === o.priority.toString())
                          .length;
```

Поскольку класс приоритета оказывается полезным в разных местах, я хочу позволить клиентам заказа использовать метод установки значения с экземпляром приоритета, что и делаю, изменяя конструктор класса приоритета.

```
class Priority...
constructor(value) {
  if (value instanceof Priority) return value;
  this._value = value;
}
```

Смысл всего этого в том, что теперь мой новый класс приоритета может быть полезен в качестве места для нового поведения — либо для нового кода, либо для кода, перенесенного из другого места. Вот простой код для добавления проверки значений приоритета и логики сравнения.

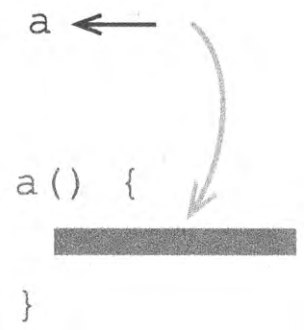
```
class Priority...
  constructor(value) {
    if (value instanceof Priority) return value;
    if (Priority.legalValues().includes(value))
      this._value = value;
    else
      throw new Error(`<${value}> is invalid for Priority`);
  }
  toString() {return this._value;}
  get _index() {
    return Priority.legalValues()
                   .findIndex(s => s === this._value);}
  static legalValues() {return ['low', 'normal', 'high', 'rush'];}

  equals(other) {return this._index === other._index;}
  higherThan(other) {return this._index > other._index;}
  lowerThan(other) {return this._index < other._index;}
```

Делая это, я решаю, что приоритет должен быть объектом-значением, а потому я предоставляю метод `equals` и обеспечиваю неизменяемость этого объекта.

Теперь, добавив поведение, я могу сделать клиентский код более выразительным:

```
Клиент...
highPriorityCount =
  orders.filter(o => o.priority.higherThan(new Priority("normal")))
                               .length;
```

Замена временной переменной запросом (Replace Temp with Query)




```
const basePrice = this._quantity * this._itemPrice;
if (basePrice > 1000)
  return basePrice * 0.95;
else
  return basePrice * 0.98;
```

```
get basePrice() {this._quantity * this._itemPrice;}

...

if (this.basePrice > 1000)
  return this.basePrice * 0.95;
else
  return this.basePrice * 0.98;
```

Мотивация

Одно из применений временных переменных — хранение некоторого вычисленного кодом значения для последующего обращения к нему в функции. Использование временной переменной позволяет мне обращаться к значению, объясняя его смысл и избегая дублирования вычисляющего его кода. Но несмотря на удобство использование переменной, зачастую *стоит* использовать вместо нее функцию.

Если я работаю над разбиением большой функции, превращение переменных в отдельные функции облегчает извлечение частей этой большой функции, поскольку мне больше не нужно передавать в извлеченные функции переменные.

Внедрение этой логики в функции зачастую также устанавливает более строгую границу между извлеченной логикой и исходной функцией, что помогает выявлять излишние зависимости и избегать их и побочных действий.

Использование функций вместо переменных позволяет также избежать дублирования логики вычислений в схожих функциях. Всякий раз, когда я вижу переменные, одинаково вычисляемые в разных местах, я стараюсь превратить их в единственную функцию.

Этот рефакторинг лучше всего работает, когда я нахожусь внутри класса, поскольку класс предоставляет общий контекст для методов, которые я извлекаю. За пределами класса в функциях верхнего уровня может быть слишком много параметров, и это сводит на нет большую часть преимуществ использования функций. Этого можно избежать с помощью вложенных функций, но они ограничивают возможности совместного использования логики связанными функциями.

Для данного рефакторинга подходят не все временные переменные. Переменная должна рассчитываться однократно, а затем только считываться. В простейшем случае это означает, что присваивание переменной выполняется только один раз. Однако возможны и несколько присваиваний в более сложном фрагменте кода — и все они должны быть извлечены в запрос. Кроме того, логика, используемая для вычисления переменной, должна давать один и тот же результат при последующем использовании переменной, что исключает применение переменных в качестве "снимков" с именами наподобие `oldAddress`.

Техника

- Убедитесь, что переменная полностью определена до ее использования, и что код, который ее вычисляет, выдает одно и то же значение при каждом ее использовании.
- Если переменная не является переменной только для чтения, но может быть сделана таковой — сделайте это.
- Выполните тестирование.
- Извлеките присваивание переменной в функцию.

  Если переменная и функция не могут совместно использовать одно и то же имя, используйте для функции временное имя.

  Убедитесь, что извлекаемая функция не имеет побочных действий. Если это не так — используйте рефакторинг *Отделение запроса от модификатора* (с. 352).
- Выполните тестирование.
- Воспользуйтесь рефакторингом *Встраивание переменной* (с. 169) для удаления временной переменной.

Пример

Вот простой класс.

```
class Order...
constructor(quantity, item) {
  this._quantity = quantity;
  this._item = item;
}

get price() {
  var basePrice = this._quantity * this._item.price;
  var discountFactor = 0.98;
  if (basePrice > 1000) discountFactor -= 0.03;
  return basePrice * discountFactor;
}
```

Я хочу заменить временные переменные `basePrice` и `discountFactor` методами.

Начиная с `basePrice`, я делаю ее константой (`const`) и выполняю тесты. Это хороший способ проверить, что я не пропустил повторное присваивание — конечно, вряд ли это возможно при такой короткой функции, но такое часто встречается, когда приходится работать с реально большим кодом.

```
class Order...
constructor(quantity, item) {
  this._quantity = quantity;
  this._item = item;
}

get price() {
  const basePrice = this._quantity * this._item.price;
  var discountFactor = 0.98;
  if (basePrice > 1000) discountFactor -= 0.03;
  return basePrice * discountFactor;
}
```

Затем я извлекаю правую часть присваивания в метод доступа.

```
class Order...
get price() {
  const basePrice = this.basePrice;
  var discountFactor = 0.98;
  if (basePrice > 1000) discountFactor -= 0.03;
  return basePrice * discountFactor;
}

get basePrice() {
  return this._quantity * this._item.price;
}
```

Выполняю тестирование и применяю рефакторинг *Встраивание переменной* (с. 169).

```
class Order...
get price() {
  const basePrice = this.basePrice;
  var discountFactor = 0.98;
  if (this.basePrice > 1000) discountFactor -= 0.03;
  return this.basePrice * discountFactor;
}
```

Затем я повторяю те же шаги с переменной `discountFactor`, начиная с применения рефакторинга *Извлечение функции* (с. 152).

```
class Order...
get price() {
  const discountFactor = this.discountFactor;
  return this.basePrice * discountFactor;
}

get discountFactor() {
  var discountFactor = 0.98;
  if (this.basePrice > 1000) discountFactor -= 0.03;
  return discountFactor;
}
```

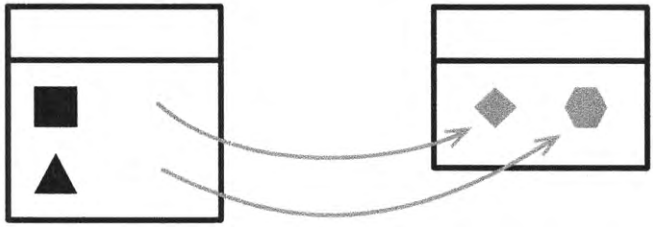
В этом случае мне нужно, чтобы моя извлеченная функция содержала оба присваивания переменной `discountFactor`. Я также могу установить исходную переменную как `const`.

Затем я выполняю встраивание.

```
get price() {
  return this.basePrice * this.discountFactor;
}
```

Извлечение класса (Extract Class)

Обратный к рефакторингу *Встраивание класса* (с. 232).



```
class Person {
  get officeAreaCode() {return this._officeAreaCode;}
  get officeNumber() {return this._officeNumber;}
```

```
class Person {
  get officeAreaCode() {return this._telephoneNumber.areaCode;}
  get officeNumber() {return this._telephoneNumber.number;}
}

class TelephoneNumber {
  get areaCode() {return this._areaCode;}
  get number() {return this._number;}
}
```

Мотивация

Вы, вероятно, сталкивались с рекомендациями, согласно которым класс должен быть четкой абстракцией, выполнять только несколько точно определенных обязанностей и так далее. На практике классы растут. Вы добавляете то операцию здесь, то немного данных там. Вы добавляете классу ответственность, чувствуя, что для нее не нужно создавать отдельный класс, но по мере роста и размножения этой ответственности класс становится слишком сложным. Вскоре становится трудно найти что-либо более далекое от термина "четкая абстракция", чем ваш класс.

Представьте себе класс с множеством методов и большим количеством данных. Класс, слишком большой, чтобы его можно было легко понять. Вам нужно подумать, как его можно разделить на части — и выполнить это разделение. Хорошим признаком для разделения служит наличие подмножества данных и подмножества методов, которые "держатся вместе". Другими хорошими признаками являются подмножества данных, которые обычно изменяются вместе или сильно

зависят одни от других. Полезный тест заключается в том, чтобы спросить себя, что произойдет, если вы удалите некоторую часть данных или метод. Какие при этом поля и методы станут бессмыслицей?

Техника

- Решите, каким образом будет разделена ответственность класса.
- Создайте новый дочерний класс для выражения разделения ответственности.

  Если ответственность исходного родительского класса больше не соответствует его имени, переименуйте родительский класс.
- При конструировании родительского класса создайте экземпляр дочернего класса и добавьте ссылку от родительского класса к дочернему.
- Примените рефакторинг *Перенос поля* (с. 253) к каждому полю, которое вы хотите перенести. Выполните тестирование после каждого перемещения.
- Используйте рефакторинг *Перенос функции* (с. 244) для перемещения методов в новый дочерний класс. Начните с низкоуровневых методов (вызываемых, а не вызывающих). Выполните тестирование после каждого перемещения.
- Просмотрите интерфейсы обоих классов, удалите ненужные методы, измените имена, чтобы они лучше соответствовали новым обстоятельствам.
- Решите, следует ли раскрывать новый дочерний класс. Если да — рассмотрите возможность применения рефакторинга *Замена ссылки значением* (с. 296) к дочернему классу.

Пример

Начну с простого класса человека (`Person`).

```
class Person...
get name() {return this._name;}
set name(arg) {this._name = arg;}
get telephoneNumber() {
  return `(${this.officeAreaCode}) ${this.officeNumber}`;}
get officeAreaCode() {return this._officeAreaCode;}
set officeAreaCode(arg) {this._officeAreaCode = arg;}
get officeNumber() {return this._officeNumber;}
set officeNumber(arg) {this._officeNumber = arg;}
```

Здесь я могу разделить выделить поведение, связанное с телефонным номером, в отдельный класс. Начну с определения пустого класса номера телефона:

```
class TelephoneNumber {
}
```

Это было просто. Далее я создаю экземпляр телефонного номера при конструировании Person.

```
class Person...
constructor() {
  this._telephoneNumber = new TelephoneNumber();
}

class TelephoneNumber...
get officeAreaCode() {return this._officeAreaCode;}
set officeAreaCode(arg) {this._officeAreaCode = arg;}
```

Затем применяю рефакторинг *Перенос поля* (с. 253) к одному из полей.

```
class Person...
get officeAreaCode() {return this._telephoneNumber.officeAreaCode;}
set officeAreaCode(arg) {this._telephoneNumber.officeAreaCode=arg;}
```

Выполняю тестирование, а затем перемещаю следующее поле.

```
class TelephoneNumber...
get officeNumber() {return this._officeNumber;}
set officeNumber(arg) {this._officeNumber = arg;}

class Person...
get officeNumber() {return this._telephoneNumber.officeNumber;}
set officeNumber(arg) {this._telephoneNumber.officeNumber = arg;}
```

Я вновь выполняю тестирование, а затем перемещаю метод телефонного номера.

```
class TelephoneNumber...
get telephoneNumber() {
  return `(${this.officeAreaCode}) ${this.officeNumber}`;}

class Person...
get telephoneNumber() {
  return this._telephoneNumber.telephoneNumber;}
```

Теперь я должен привести все в порядок. Наличие "офиса" как части кода телефонного номера смысла не имеет, поэтому я выполняю переименование.

```
class TelephoneNumber...
get areaCode() {return this._areaCode;}
set areaCode(arg) {this._areaCode = arg;}

get number() {return this._number;}
set number(arg) {this._number = arg;}

class Person...
get officeAreaCode() {return this._telephoneNumber.areaCode;}
set officeAreaCode(arg) {this._telephoneNumber.areaCode = arg;}
get officeNumber() {return this._telephoneNumber.number;}
set officeNumber(arg) {this._telephoneNumber.number = arg;}
```

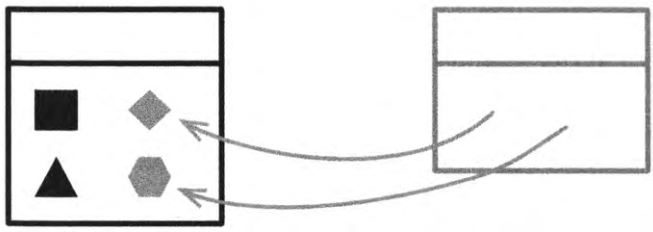
Метод телефонного номера в классе телефонного номера также не имеет особого смысла, так что я применяю рефакторинг *Изменение объявления функции* (с. 170).

```
class TelephoneNumber...
toString() {return `(${this.areaCode}) ${this.number}`;}

class Person...
get telephoneNumber() {return this._telephoneNumber.toString();}
```

Телефонные номера, как правило, полезны, поэтому я думаю, что открою новый объект клиентам. Я могу заменить эти "офисные" методы методами доступа к телефонному номеру. Но в этом случае телефонный номер будет работать лучше в качестве объекта-значения [39], поэтому я сначала применил бы рефакторинг *Замена ссылки значением* (с. 296) (пример к этому рефакторингу показывает, как это делается для телефонного номера).

Встраивание класса (Inline Class)

Обратный к рефакторингу *Извлечение класса* (с. 229).



```
class Person {
  get officeAreaCode() {return this._telephoneNumber.areaCode;}
  get officeNumber() {return this._telephoneNumber.number;}
}
class TelephoneNumber {
  get areaCode() {return this._areaCode;}
  get number() {return this._number;}
}
```

```
class Person {
  get officeAreaCode() {return this._officeAreaCode;}
  get officeNumber() {return this._officeNumber;}
```

Мотивация

Данный рефакторинг является обратным к рефакторингу *Извлечение класса* (с. 229). Я использую его, если класс невелик и больше не должен быть отдельным классом. Зачастую это результат рефакторинга, который переносит из класса другую ответственность, так что у него остается немного обязанностей. В этот момент я переношу класс в другой — тот, который больше всего использует наш микрокласс.

Другая причина использовать данный рефакторинг — если у меня есть два класса, которые я хочу преобразовать в пару классов с различным распределением функциональных возможностей. Возможно, проще всего будет сначала использовать рефакторинг *Встраивание класса*, чтобы объединить их в один класс, а уже затем прибегнуть к рефакторингу *Извлечение класса* (с. 229), чтобы создать новое разделение. Это распространенный при реорганизации подход: иногда проще перемещать элементы по одному из одного контекста в другой, а иногда лучше сначала собрать контексты воедино, и уже затем использовать рефакторинг извлечения для нового разделения единого целого на разные элементы.

Техника

- Создайте в целевом классе функции для всех открытых функций исходного класса. Эти функции должны просто выполнять делегирование исходному классу.
- Измените все обращения к методам исходного класса таким образом, чтобы вместо них использовались делегаты целевого класса. Выполните тестирование после каждого изменения.
- Переместите все функции и данные из исходного класса в целевой, выполняя тестирование после каждого перемещения, пока исходный класс не станет пустым.
- Удалите исходный класс и проведите короткую, простую похоронную службу.

Пример

Вот класс, который содержит несколько фрагментов сопроводительной информации о доставке.

```
class TrackingInformation {
  get shippingCompany() {return this._shippingCompany;}
  set shippingCompany(arg) {this._shippingCompany = arg;}
  get trackingNumber() {return this._trackingNumber;}
  set trackingNumber(arg) {this._trackingNumber = arg;}
```

```
  get display() {
    return `${this.shippingCompany}: ${this.trackingNumber}`;
  }
}
```

Он используется как часть класса поставки.

```
class Shipment...
get trackingInfo() {
  return this._trackingInformation.display;
}
get trackingInformation() {return this._trackingInformation;}
set trackingInformation(aTrackingInformation) {
  this._trackingInformation = aTrackingInformation;
}
```

Хотя этот класс, возможно, был полезен в прошлом, я больше не вижу в нем необходимости, и хочу включить его в `Shipment`.

Начну с поиска мест вызова методов `TrackingInformation`.

```
Точка вызова...
aShipment.trackingInformation.shippingCompany = request.vendor;
```

Я собираюсь переместить все такие функции в `Shipment`, но делаю это немного иначе, чем обычно делает рефакторинг *Перенос функции* (с. 244). В данном случае я начну с добавления метода делегирования в `Shipment` и настройки клиента для его вызова.

```
class Shipment...
set shippingCompany(arg) {
  this._trackingInformation.shippingCompany = arg;}

Точка вызова...
aShipment.~~trackingInformation~~.shippingCompany = request.vendor;
```

Я делаю это для всех элементов сопроводительной информации, которые используются клиентами. После этого я могу переместить все эти элементы в класс `Shipment`.

Начну с применения рефакторинга *Встраивание функции* (с. 161) к методу `display`.

```
class Shipment...
get trackingInfo() {
  return `${this.shippingCompany}: ${this.trackingNumber}`;
}
```

Далее я перемещаю поле транспортной компании.

```
get shippingCompany() {
  return this.~~_trackingInformation~~._shippingCompany;}
set shippingCompany(arg) {
  this.~~_trackingInformation~~._shippingCompany = arg;}
```

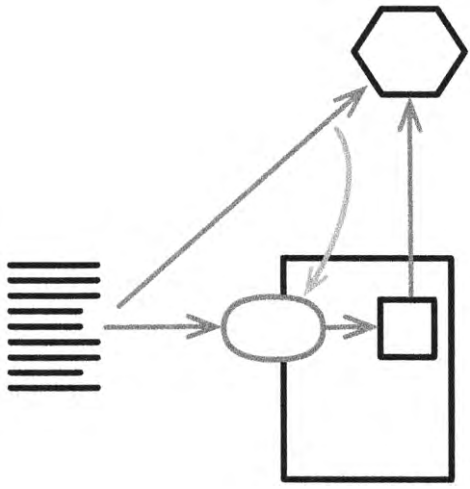
Я не использую полную технику рефакторинга *Перенос поля* (с. 253), так как в данном случае обращаюсь только к полю `shippingCompany` из `Shipment`, которое и является целью перемещения. Таким образом, мне не нужны шаги, которые перемещают обращение из исходного объекта в целевой.

Продолжаю, пока не будет перенесено все необходимое. После этого я могу удалить класс сопроводительной информации.

```
class Shipment...
get trackingInfo() {
  return `${this.shippingCompany}: ${this.trackingNumber}`;
}
get shippingCompany()    {return this._shippingCompany;}
set shippingCompany(arg) {this._shippingCompany = arg;}
get trackingNumber()     {return this._trackingNumber;}
set trackingNumber(arg)  {this._trackingNumber = arg;}
```

Сокрытие делегата (Hide Delegate)

Обратный к рефакторингу *Удаление посредника* (с. 238).



```
manager = aPerson.department.manager;
```

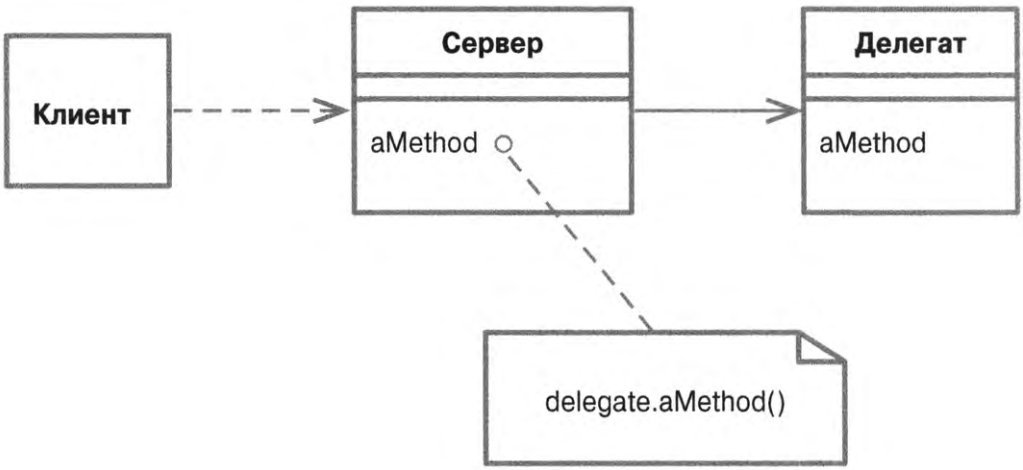
```
manager = aPerson.manager;

class Person {
  get manager() {return this.department.manager;}
```

Мотивация

Одним из ключей — если не единственным ключом — к хорошему модульному проекту является инкапсуляция. Инкапсуляция означает, что модули должны знать о других частях системы как можно меньше. Тогда, если что-то в программе меняется, об этом нужно рассказать только малому числу модулей, что облегчает внесение изменений.

Когда мы впервые узнаем об объектно-ориентированном программировании, нам говорят, что инкапсуляция означает сокрытие наших полей. По мере того как мы становимся все более опытными программистами, мы понимаем, что это — не единственное, что можно инкапсулировать.

Если у меня есть некоторый клиентский код, который вызывает метод, определенный для объекта в поле серверного объекта, то клиент должен знать об этом объекте-делегате. Если делегат меняет свой интерфейс, изменения распространяются на всех клиентов сервера, которые используют делегат. Я могу убрать эту зависимость, разместив на сервере простой метод делегирования, который скрывает делегат. Тогда любые изменения, внесенные в делегат, будут распространяться только на сервер, и не будут затрагивать клиентов.



Техника

- Для каждого метода делегата создайте простой делегирующий метод на сервере.
- Настройте клиент для вызова сервера. Выполняйте тестирование после каждого изменения.

- Если больше нет клиентов, которым требуется доступ к делегату, удалите серверный метод доступа к делегату.
- Выполните тестирование.

Пример

Начну с классов `Person` и `Department`.

```
class Person...
constructor(name) {
  this._name = name;
}
get name() {return this._name;}
get department()    {return this._department;}
set department(arg) {this._department = arg;}

class Department...
get chargeCode()    {return this._chargeCode;}
set chargeCode(arg) {this._chargeCode = arg;}
get manager()    {return this._manager;}
set manager(arg) {this._manager = arg;}
```

Некоторый клиентский код хочет узнать руководителя определенного человека. Для этого нужно сначала определить его отдел.

```
Код клиента...
manager = aPerson.department.manager;
```

Это раскрывает клиенту, как работает класс отдела, и что именно отдел отвечает за отслеживание менеджера. Я могу уменьшить связывание, скрывая класс отдела от клиента. Я делаю это, создавая простой метод делегирования в классе `Person`:

```
class Person...
get manager() {return this._department.manager;}
```

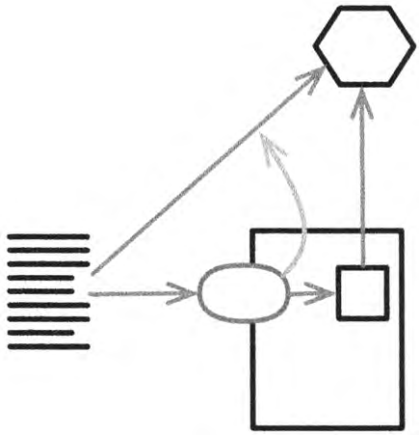
Теперь мне нужно изменить всех клиентов класса `Person` так, чтобы они использовали новый метод:

```
Код клиента...
manager = aPerson.department.manager;
```

Внеся изменения во все методы класса отдела и для всех клиентов класса `Person`, я могу удалить метод доступа `department` в классе `Person`.

Удаление посредника (Remove Middle Man)

Обратный к рефакторингу *Сокрытие делегата* (с. 235).



```
manager = aPerson.manager;

class Person {
get manager() {return this.department.manager;}
```

```
manager = aPerson.department.manager;
```

Мотивация

В мотивации рефакторинга *Сокрытие делегата* (с. 235) я писал о преимуществах инкапсуляции использования делегирования. У всего есть своя цена. Каждый раз, когда клиент хочет использовать новую функциональную возможность делегата, я должен добавить простой метод делегирования на сервер. После ряда добавлений функциональных возможностей такая пересылка начинает раздражать. Серверный класс оказывается просто посредником (см. запах *Посредник* в главе 3), и, возможно, пришло время клиенту работать с делегатом непосредственно.

Трудно точно определить, какое количество сокрытия является правильным. К счастью, при наличии рефакторинга *Сокрытие делегата* (с. 235) и описываемого в этом разделе это не так важно. Я могу в любой момент скорректировать свой код. По мере изменения системы меняется и количество сокрытия. Инкапсуляция, бывшая хорошей полгода назад, сегодня может стать неудобной. Рефакторинг означает, что мне никогда не придется жалеть о сделанном — я просто исправляю его.

Техника

- Создайте метод доступа для делегата.
- Для каждого клиента, использующего делегирующий метод, замените вызов этого метода цепочкой с использованием метода доступа. Выполните тестирование после каждой замены.

  Если все вызовы делегирующего метода заменены, вы можете удалить делегирующий метод.

  При использовании автоматического рефакторинга можете применить рефакторинг *Инкапсуляция переменной* (с. 178) к полю делегата, а затем — рефакторинг *Встраивание функции* (с. 161) для всех методов, которые используют эту функцию.

Пример

Я начну с класса человека `Person`, который использует связанный объект отдела для определения менеджера. (Если вы читаете эту книгу последовательно, этот пример может показаться вам очень знакомым.)

```
Код клиента...
manager = aPerson.manager;

class Person...
get manager() {return this._department.manager;}

class Department...
get manager() {return this._manager;}
```

Этот подход прост в использовании и инкапсулирует отдел. Однако, если это делают многие методы, я получаю слишком большое количество таких простых делегирований в классе `Person`. В этом случае имеет смысл удаление посредника. Сначала я создаю метод доступа для делегата:

```
class Person...
get department() {return this._department;}
```

Теперь я по очереди иду к каждому клиенту и модифицирую его так, чтобы класс отдела использовался им непосредственно.

```
Код клиента...
manager = aPerson.department.manager;
```

Сделав это со всеми клиентами, я могу удалить метод `manager` из класса `Person`. Я могу повторить этот процесс для любых других простых делегирований класса `Person`.

Можно смешать два подхода. Некоторые делегирования могут быть настолько распространены, что их желательно оставить для облегчения работы с клиентским кодом. Нет никакой причины, по которой я должен либо скрывать делегата, либо удалять посредника — конкретные обстоятельства сами определяют верный выбор подхода, и разработчики могут по-разному решать, какой именно подход наиболее хорош в данном конкретном случае.

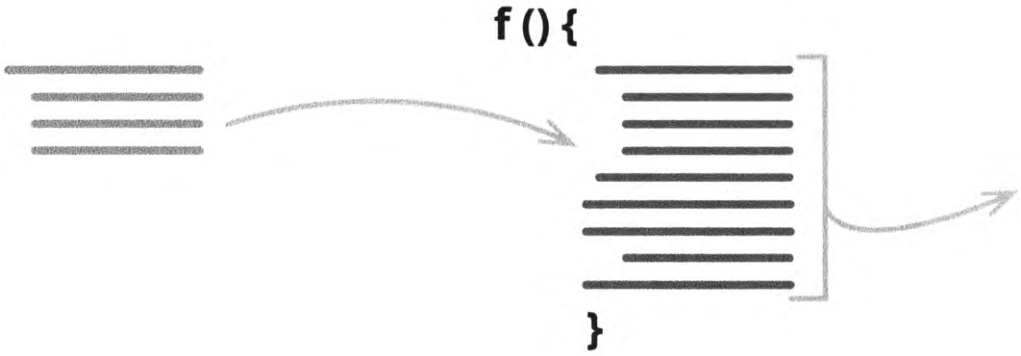
Если у меня есть автоматические рефакторинги, то имеется полезная версия для описанных шагов. Сначала я применяю рефакторинг *Инкапсуляция переменной* (с. 178) к `department`. Этот рефакторинг изменяет метод доступа к менеджеру таким образом, чтобы использовать открытый метод доступа к отделу:

```
class Person...
get manager() {return this.department.manager;}
```

Изменение в JavaScript очень тонкое, но, удалив подчеркивание из `department`*, я использую новый метод доступа, а не непосредственный доступ к полю.*

Затем я применяю рефакторинг *Встраивание функции* (с. 161) к методу `manager`, чтобы выполнить замену всего вызывающего кода за один раз.

Подстановка алгоритма (Substitute Algorithm)




```
function foundPerson(people) {
  for(let i = 0; i < people.length; i++) {
    if (people[i] === "Don") {
      return "Don";
    }
    if (people[i] === "John") {
      return "John";
    }
    if (people[i] === "Kent") {
      return "Kent";
```

```
    }
  }
  return "";
}
```

```
function foundPerson(people) {
  const candidates = ["Don", "John", "Kent"];
  return people.find(p => candidates.includes(p)) || '';
}
```

Мотивация

Почти всегда свет не сходится клином на единственном способе достижения цели. Как правило, есть несколько способов добиться того или иного. Я уверен, что некоторые из этих способов легче других. Так же и с алгоритмами. Если я нахожу более простой и понятный способ сделать что-то, я заменяю сложный способ более простым. Рефакторинг может разбить нечто сложное на более простые части, но иногда я просто достигаю точки, в которой мне нужно удалить весь алгоритм и заменить его чем-то более простым. Это происходит, когда я узнаю о проблеме больше и понимаю, что есть более простой способ достичь нужного результата. Это происходит и тогда, когда я начинаю использовать библиотеку, предоставляющую функции, дублирующие мой код.

Иногда, когда я хочу изменить алгоритм, чтобы он работал немного по-другому, проще начать с замены его чем-то, что сделало бы мое изменение более простым для выполнения.

Когда я вынужден делать этот шаг, я должен быть уверен, что разделил метод на части настолько, насколько это можно. Замена большого, сложного алгоритма очень трудна; только деля его на более простые части, я смогу успешно справиться со стоящей передо мной задачей.

Техника

- Организуйте заменяемый код так, чтобы он выполнял всю функцию полностью.
- Подготовьте тесты, применяя только эту функцию, чтобы зафиксировать ее поведение.
- Подготовьте альтернативный алгоритм.
- Выполните статические проверки.
- Выполните тесты для сравнения выходных данных старого алгоритма с новым. Если они одинаковы, все в порядке. В противном случае используйте старый алгоритм для сравнения при тестировании и отладке.

Глава 8

Перенос функциональности

До сих пор рефакторинги были посвящены созданию, удалению и переименованию элементов программы. Другой важной частью рефакторинга является перемещение элементов между контекстами. Я использую рефакторинг *Перенос функции* (с. 244) для перемещения функций между классами и другими модулями. Перемещать можно и поля — с помощью рефакторинга *Перенос поля* (с. 253).

Можно перемещать и отдельные инструкции. Рефакторинги *Перенос инструкций в функцию* (с. 258) и *Перенос инструкций в точку вызова* (с. 262) позволяют вносить их в функции или извлекать оттуда, а рефакторинг *Перемещение инструкций* (с. 269) обеспечивает перемещение инструкций в пределах функции. Иногда при наличии инструкций, дублирующих код некоторой существующей функции, такое дублирование можно удалить с помощью рефакторинга *Замена встроенного кода вызовом функции* (с. 268).

Я часто прибегаю к двум рефакторингам над циклами: *Разделение цикла* (с. 274), чтобы гарантировать, что цикл решает только одну задачу, и *Замена цикла конвейером* (с. 278), чтобы полностью избавиться от цикла.

Есть еще один любимый рефакторинг многих хороших программистов — *Удаление неработающего кода* (с. 283). Ничто не доставляет такого удовольствия, как применение цифрового огнемета к лишним инструкциям.

Перенос функции (Move Function)

Бывший рефакторинг *Перенос метода*

```
class Account {
  get overdraftCharge() {...}
```

```
class AccountType {
  get overdraftCharge() {...}
```

Мотивация

Сердцем хорошего проекта программного обеспечения является его модульность, которая обеспечивает возможность вносить множество изменений в программу, в то же время понимая лишь небольшую ее часть. Чтобы добиться этой модульности, я должен убедиться, что связанные программные элементы сгруппированы и связи между ними легко найти и понять. Но мое понимание того, как это сделать, не является статичным — со временем, начиная лучше разбираться в том, что я делаю, я понимаю, как наилучшим образом сгруппировать элементы программного обеспечения. Чтобы отразить это растущее понимание, мне нужно иметь возможность переносить элементы программы.

Все функции живут в некотором контексте; он может быть глобальным, но обычно это некоторая разновидность модуля. В объектно-ориентированной программе основной модульный контекст — класс. Вложение функции в другую создает другой распространенный контекст. Различные языки предоставляют различные формы модульности, каждая из которых создает контекст для "проживания" функции.

Одна из самых простых причин для переноса функций — когда она обращается к элементам в других контекстах больше, чем к элементам контекста, в котором она находится в настоящее время. Ее перенос вместе с этими элементами часто улучшает инкапсуляцию, позволяя другим частям программного обеспечения меньше зависеть от деталей данного модуля.

Точно так же я могу перенести функцию из-за местоположений ее вызовов или из-за того, откуда она будет вызываться при планируемом усовершенствовании.

Функция, определенная как вспомогательная внутри другой функции, может быть ценной и сама по себе, а потому ее стоит перенести в более доступное место. Метод класса может оказаться проще использовать, если он будет перенесен в другой класс.

Решение о переносе функции редко бывает легким. Чтобы помочь себе принять его, я проверяю текущий и потенциальный контексты функции. Мне нужно посмотреть, какие функции вызывают данную, какие функции вызывает переносимая функция и какие данные ею используются. Часто я вижу, что мне нужен новый контекст для группы функций, и я создаю его с помощью рефакторингов *Объединение функций в класс* (с. 190) или *Извлечение класса* (с. 229). Хотя решить, где лучше всего расположить функцию, может быть трудно — чем сложнее этот выбор, тем меньшее значение он имеет. Можете попробовать работать с функциями в том или ином контексте, осознавая, что это просто проверка, насколько хорошо он подходит, и, если это неподходящий контекст — всегда можно перенести их позже.

Техника

- Изучите все элементы программы, используемые выбранной функцией в ее текущем контексте. Подумайте, не следует ли перенести и их тоже.

  Обнаружив вызываемую функцию, которая также должна быть перенесена, я обычно перемещаю ее первой. Таким образом, перемещение группы функций начинается с той, которая меньше всего зависит от других в группе.

  Если высокоуровневая функция является единственной, вызывающей подфункции, то можно встроить эти функции в высокоуровневый метод, переместить его и повторно извлечь их в месте назначения.

- Проверьте, не является ли выбранная функция полиморфным методом.

  Используя объектно-ориентированный язык, я должен учитывать объявления родительских и дочерних классов.

- Скопируйте функцию в целевой контекст. Настройте ее так, чтобы она соответствовала своему "новому дому".

  Если тело функции использует элементы исходного контекста, мне нужно либо передавать эти элементы в качестве параметров, либо передать ссылку на этот исходный контекст.

  Перенос функции часто означает, что мне нужно придумать ей другое имя, которое лучше соответствует новому контексту.

- Выполните статический анализ.

- Выясните, как обращаться к целевой функции из исходного контекста.
- Превратите исходную функцию в делегирующую.
- Выполните тестирование.
- Рассмотрите возможность применения рефакторинга *Встраивание функции* (с. 161) к исходной функции.

  Исходная функция может оставаться делегирующей функцией. Но если вызывающие ее функции могут легко обратиться к целевой функции непосредственно — лучше удалить посредника.

Пример: перенос вложенной функции на верхний уровень

Я начну с функции, которая рассчитывает общее расстояние для записи трека GPS.

```
function trackSummary(points) {
  const totalTime = calculateTime();
  const totalDistance = calculateDistance();
  const pace = totalTime / 60 / totalDistance ;
  return {
      time:     totalTime,
      distance: totalDistance,
      pace:     pace
  };

  function calculateDistance() {
    let result = 0;
    for (let i = 1; i < points.length; i++) {
      result += distance(points[i-1], points[i]);
    }
    return result;
  }

  function distance(p1,p2) { ... }
  function radians(degrees) { ... }
  function calculateTime() { ... }
}
```

Я бы хотел переместить `calculateDistance` на верхний уровень, чтобы можно было рассчитывать расстояния для треков без всех других частей итоговой информации.

Начнем с копирования функции на верхний уровень.

```
function trackSummary(points) {
  const totalTime = calculateTime();
  const totalDistance = calculateDistance();
  const pace = totalTime / 60 / totalDistance ;
  return {
```

```
  time:     totalTime,
  distance: totalDistance,
  pace:     pace
  };

  function calculateDistance() {
    let result = 0;
    for (let i = 1; i < points.length; i++) {
      result += distance(points[i-1], points[i]);
    }
    return result;
  }
  ...

  function distance(p1,p2) { ... }
  function radians(degrees) { ... }
  function calculateTime() { ... }
}

function top_calculateDistance() {
  let result = 0;
  for (let i = 1; i < points.length; i++) {
    result += distance(points[i-1], points[i]);
  }
  return result;
}
```

Копируя такую функцию, я предпочитаю изменять ее имя, чтобы можно было различать эти копии как в коде, так и в голове. Прямо сейчас я не хочу думать о том, какое имя должно быть правильным, так что я создаю просто временное имя.

Программа все еще работоспособна, но статический анализ справедливо указывает на неприятности. Новая функция имеет два неопределенных символа: `distance` и `points`. Естественный способ работы с `points` — передача в качестве параметра.

```
function top_calculateDistance(points) {
  let result = 0;
  for (let i = 1; i < points.length; i++) {
    result += distance(points[i-1], points[i]);
  }
  return result;
}
```

Я мог бы точно так же поступить и с `distance`, но, возможно, имеет смысл перенести эту переменную вместе с `calculateDistance`. Вот как выглядит соответствующий код.

```
function trackSummary...
function distance(p1,p2) {
  // См. формулу http://www.movable-type.co.uk/scripts/latlong.html
```

```
  const EARTH_RADIUS = 3959; // in miles
  const dLat = radians(p2.lat) - radians(p1.lat);
  const dLon = radians(p2.lon) - radians(p1.lon);
  const a = Math.pow(Math.sin(dLat / 2),2)
          + Math.cos(radians(p2.lat))
          * Math.cos(radians(p1.lat))
          * Math.pow(Math.sin(dLon / 2), 2);
  const c = 2 * Math.atan2(Math.sqrt(a), Math.sqrt(1-a));
  return EARTH_RADIUS * c;
}

function radians(degrees) {
  return degrees * Math.PI / 180;
}
```

Я вижу, что `distance` использует только `radians`, а `radians` в своем текущем контексте не использует ничего. Поэтому вместо передачи функций можно перенести и эту функцию тоже. Я могу сделать небольшой шаг в этом направлении, переместив ее из текущего контекста, и вложив в `calculateDistance`.

```
function trackSummary(points) {
  const totalTime = calculateTime();
  const totalDistance = calculateDistance();
  const pace = totalTime / 60 / totalDistance ;
  return {
      time:     totalTime,
      distance: totalDistance,
      pace:     pace
  };

  function calculateDistance() {
    let result = 0;
    for (let i = 1; i < points.length; i++) {
      result += distance(points[i-1], points[i]);
    }
    return result;

    function distance(p1,p2) { ... }
    function radians(degrees) { ... }
  }
```

Поступив таким образом, я могу использовать как статический анализ, так и тестирование, чтобы выяснить, нет ли каких-либо осложнений. В данном случае все в порядке, поэтому я могу скопировать их в `top_calculateDistance`.

```
function top_calculateDistance(points) {
  let result = 0;
  for (let i = 1; i < points.length; i++) {
    result += distance(points[i-1], points[i]);
  }
  return result;
```

```
  function distance(p1,p2) { ... }
  function radians(degrees) { ... }
}
```

Копирование не меняет способ работы программы, но дает возможность для большего статического анализа. Если бы я не заметил, что `distance` вызывает `radians`, то это было бы обнаружено инструментами статического анализа.

Теперь, когда я все подготовил для основных изменений, пришло их время — тело исходного метода `calculateDistance` теперь будет вызывать `top_calculateDistance`.

```
function trackSummary(points) {
  const totalTime = calculateTime();
  const totalDistance = calculateDistance();
  const pace = totalTime / 60 / totalDistance ;
  return {
      time:     totalTime,
      distance: totalDistance,
      pace:     pace
  };

  function calculateDistance() {
    return top_calculateDistance(points);
  }
```

Наступило решающее время для выполнения тестирования, чтобы полностью убедиться, что функция успешно переехала в свой новый дом.

Действия после этого напоминают распаковку коробок с вещами после переезда. Первым делом нужно решить, сохранять исходную функцию, которая занимается только делегированием, или нет. В этом случае мест ее вызова немного, и, как обычно бывает с вложенными функциями, они сильно локализованы. Поэтому я буду рад избавиться от этого хлама.

```
function trackSummary(points) {
  const totalTime = calculateTime();
  const totalDistance = top_calculateDistance(points);
  const pace = totalTime / 60 / totalDistance ;
  return {
      time:     totalTime,
      distance: totalDistance,
      pace:     pace
  };
```

Сейчас также самое время подумать над именем. Поскольку функция верхнего уровня имеет наивысшую видимость, мне бы хотелось, чтобы у нее было имя получше. Хорошим выбором кажется `totalDistance`. Я не могу использовать его немедленно, так как оно будет скрыто переменной в `trackSummary`, но не вижу никакой причины его сохранять в любом случае, так что применяю к нему рефакторинг *Встраивание переменной* (с. 169).

```
function trackSummary(points) {
  const totalTime = calculateTime();
  const pace = totalTime / 60 / totalDistance(points);
  return {
      time:     totalTime,
      distance: totalDistance(points),
      pace:     pace
  };

  function totalDistance(points) {
  let result = 0;
  for (let i = 1; i < points.length; i++) {
    result += distance(points[i-1], points[i]);
  }
  return result;
```

Если бы мне нужно было сохранить переменную, я бы переименовал ее во что-то вроде `totalDistanceCache` или `distance`.

Поскольку функции `distance` и `radians` не зависят от чего-либо внутри `totalDistance`, я предпочитаю также перенести их на верхний уровень, поместив там все четыре функции.

```
function trackSummary(points)  { ... }
function totalDistance(points) { ... }
function distance(p1,p2)       { ... }
function radians(degrees)      { ... }
```

Некоторые программисты предпочли бы оставить `distance` и `radians` внутри `totalDistance`, чтобы ограничить их видимость. В некоторых языках это может быть важным, но JavaScript ES 2015 имеет отличный модульный механизм, который является лучшим инструментом для управления видимостью функций. В целом я опасаюсь вложенных функций — они слишком легко устанавливают скрытые взаимосвязи данных, за которыми трудно уследить.

Пример: перенос между классами

Чтобы проиллюстрировать разнообразие переноса функций, в этот раз я начну со следующего кода.

```
class Account...
get bankCharge() {
  let result = 4.5;
  if (this._daysOverdrawn > 0) result += this.overdraftCharge;
  return result;
}

get overdraftCharge() {
  if (this.type.isPremium) {
    const baseCharge = 10;
```

```
    if (this.daysOverdrawn <= 7)
      return baseCharge;
    else
      return baseCharge + (this.daysOverdrawn - 7) * 0.85;
  }
  else
    return this.daysOverdrawn * 1.75;
}
```

Далее ожидаются изменения, которые приводят к тому, что разные типы счетов будут иметь разные алгоритмы определения начислений. Таким образом, кажется естественным переместить `overdraftCharge` в класс типа учетной записи.

Первый шаг состоит в том, чтобы рассмотреть функциональные возможности, используемые методом `overdraftCharge`, и выяснить, не стоит ли выполнить перенос сразу пакета методов. В этом случае нужно, чтобы метод `daysOverdrawn` оставался в классе счета, так как он меняется для разных отдельных счетов.

Затем я копирую тело метода в тип учетной записи и подгоняю его под себя.

```
class AccountType...
overdraftCharge(daysOverdrawn) {
  if (this.isPremium) {
    const baseCharge = 10;
    if (daysOverdrawn <= 7)
      return baseCharge;
    else
      return baseCharge + (daysOverdrawn - 7) * 0.85;
  }
  else
    return daysOverdrawn * 1.75;
}
```

Чтобы метод хорошо работал на новом месте, приходится иметь дело с двумя целевыми вызовами, которые изменили свою область видимости. Теперь `isPremium` — простой вызов `this`. Но с `daysOverdrawn` я должен решить — передавать значение или передавать счет? В данный момент я передаю простое значение, но вполне могу изменить этот подход в будущем, если мне потребуется нечто большее, чем просто дни (в особенности если необходимая информация будет зависеть от типа счета).

Затем я заменяю тело исходного метода делегирующим вызовом.

```
class Account...
get bankCharge() {
  let result = 4.5;
  if (this._daysOverdrawn > 0) result += this.overdraftCharge;
    return result;
  }

get overdraftCharge() {
  return this.type.overdraftCharge(this.daysOverdrawn);
}
```

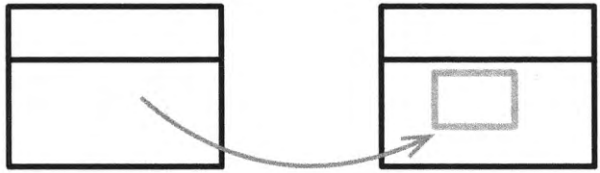
Принимаем решение о том, следует оставить делегирование как есть или выполнить встраивание `overdraftCharge`. Вот как выглядит результат встраивания.

```
class Account...
get bankCharge() {
  let result = 4.5;
  if (this._daysOverdrawn > 0)
    result += this.type.overdraftCharge(this.daysOverdrawn);
  return result;
}
```

На предыдущих этапах я передавал `daysOverdrawn` в качестве параметра, но если необходима передача большого количества данных из учетной записи, то, возможно, стоит предпочесть передачу самой учетной записи.

```
class Account...
get bankCharge() {
  let result = 4.5;
  if (this._daysOverdrawn > 0) result += this.overdraftCharge;
  return result;
}

get overdraftCharge() {
  return this.type.overdraftCharge(this);
}

class AccountType...
overdraftCharge(account) {
  if (this.isPremium) {
    const baseCharge = 10;
    if (account.daysOverdrawn <= 7)
      return baseCharge;
    else
      return baseCharge + (account.daysOverdrawn - 7) * 0.85;
  }
  else
    return account.daysOverdrawn * 1.75;
}
```

Перенос поля (Move Field)



```
class Customer {
  get plan() {return this._plan;}
  get discountRate() {return this._discountRate;}
```

```
class Customer {
  get plan() {return this._plan;}
  get discountRate() {return this.plan.discountRate;}
```

Мотивация

Программирование включает в себя написание большого количества кода, реализующего поведение, — но настоящая сила программы основана на ее структурах данных. Если у меня хороший набор структур данных, соответствующих поставленной задаче, то реализующий поведение код прост и понятен. Плохие структуры данных приводят к большому количеству кода, работа которого заключается просто в обработке плохих данных. И это не просто сложный код, который трудно понять; это также означает, что структуры данных скрывают то, что делает программа.

Итак, структуры данных важны — но, как и большинство аспектов программирования, их трудно правильно применять. При первоначальном анализе, выявляя наилучшим образом подходящие для решения задачи структуры данных, я обнаружил, что опыт и такие методы, как проектирование на основе предметной области, улучшили мою способность к анализу. Но несмотря на все мои навыки и опыт, я все же часто делаю ошибки при таком первоначальном проектировании. В процессе работы я узнаю о проблемной области и своих структурах данных все больше и больше. Разумное и правильное проектное решение через неделю может выглядеть ошибочным (и быть таковым).

Как только я осознаю, что структура данных не подходит для данной задачи — ее нужно изменить. Если я оставлю имеющиеся структуры данных со всеми их недостатками, эти недостатки будут меня запутывать и усложнять будущий код.

Я могу попытаться перенести данные, потому что мне нужно передавать поле из одной записи каждый раз, когда я передаю в функцию другую запись. Те части

данных, которые всегда передаются в функцию вместе, обычно лучше всего помещать в одну запись, чтобы прояснить их взаимосвязь. Изменение также является подобным фактором: если изменение в одной записи приводит к изменению поля в другой, то это признак того, что поле находится в неправильном месте. Если мне нужно обновлять одно и то же поле в нескольких структурах, это также признак того, что его нужно перенести в другое место, где его придется обновлять только один раз.

Обычно я выполняю данный рефакторинг в контексте более широкого набора изменений. Например, перенеся поле, я обнаруживаю, что многим пользователям этого поля лучше получить доступ к этим данным через целевой объект, а не из исходного источника, так что я вношу соответствующие изменения с помощью рефакторинга. Точно так же можно обнаружить, что в данный момент у меня не получится выполнить рефакторинг *Перенос поля* из-за способа использования данных и что сначала нужно изменить некоторые схемы этого использования, а уже затем выполнять перенос.

До сих пор в своем описании я употреблял термин *запись* (record), но все сказанное верно и для классов, и для объектов. Класс — это тип записи с присоединенными функциями, и их необходимо поддерживать в "здоровом" состоянии так же, как и любые другие данные. Функции класса облегчают перенос данных, так как последние инкапсулированы методами доступа. Я могу переносить данные, менять методы доступа, но код клиента по-прежнему будут работать. Таким образом, это рефакторинг, который легче выполнить, если у вас есть классы, и мое описание подтверждает это предположение. Если я использую "голые" записи, которые не поддерживают инкапсуляцию, я все равно смогу внести подобное изменение, но оно будет куда более сложным.

Техника

- Обеспечьте инкапсуляцию исходного поля.
- Выполните тестирование.
- Создайте поле (и методы доступа) в месте назначения.
- Выполните статические проверки.
- Обеспечьте наличие ссылки из исходного объекта к целевому объекту.

  Существующее поле или метод могут указать вам целевой объект. Если нет — посмотрите, не сможете ли вы легко создать метод, который это сделает. В противном случае вам может понадобиться создать новое поле в исходном объекте, в котором может храниться целевой объект. Это может быть как постоянным изменением, так и временным — пока вы не выполните достаточный рефакторинг в более широком контексте.

- Настройте методы доступа для использования целевого поля.

  Если целевой объект совместно используется несколькими исходными объектами, сначала рассмотрите возможность обновить метод установки значения так, чтобы изменять и целевое поле, и поле источника, после чего выполнить рефакторинг *Введение утверждения* (с. 346) для обнаружения несогласованных обновлений. Как только вы определили, что все в порядке, — завершите изменение методов доступа так, чтобы они использовали целевое поле.
- Выполните тестирование.
- Удалите исходное поле.
- Выполните тестирование.

Пример

Начну работу с классами клиента и контракта.

```
class Customer...
constructor(name, discountRate) {
  this._name = name;
  this._discountRate = discountRate;
  this._contract = new CustomerContract(dateToday());
}
get discountRate() {return this._discountRate;}
becomePreferred() {
  this._discountRate += 0.03;
  // Прочие вычисления
}
applyDiscount(amount) {
  return amount.subtract(amount.multiply(this._discountRate));
}

class CustomerContract...
constructor(startDate) {
  this._startDate = startDate;
}
```

Я хочу переместить поле учетной ставки от клиента к контракту.

Первое, что мне нужно сделать, — это использовать рефакторинг *Инкапсуляция переменной* (с. 178) для инкапсуляции доступа к полю учетной ставки.

```
class Customer...
constructor(name, discountRate) {
  this._name = name;
  this._setDiscountRate(discountRate);
  this._contract = new CustomerContract(dateToday());
}
```

```
get discountRate() {return this._discountRate;}
_setDiscountRate(aNumber) {this._discountRate = aNumber;}
becomePreferred() {
  this._setDiscountRate(this.discountRate + 0.03);
  // Прочие вычисления
}
applyDiscount(amount) {
return amount.subtract(amount.multiply(this.discountRate));
}
```

Для обновления учетной ставки я использую метод, а не истинное свойство, так как не хочу создавать для нее открытый метод установки значения.

Добавляю поле и методы доступа к классу контракта.

```
class CustomerContract...
constructor(startDate, discountRate) {
  this._startDate = startDate;
  this._discountRate = discountRate;
}
get discountRate() {return this._discountRate;}
set discountRate(arg) {this._discountRate = arg;}
```

Теперь я изменяю методы доступа клиента, чтобы использовать новое поле. Когда я сделал это, то получил сообщение об ошибке: "Невозможно установить свойство `discountRate` неопределенного класса". Это произошло потому, что `_setDiscountRate` был вызван до того, как я создал объект контракта в конструкторе. Чтобы исправить ситуацию, я сначала вернулся к предыдущему состоянию, а затем использовал рефакторинг *Перемещение инструкций* (с. 269), чтобы поместить `_setDiscountRate` после создания контракта.

```
class Customer...
constructor(name, discountRate) {
  this._name = name;
  this._setDiscountRate(discountRate);
  this._contract = new CustomerContract(dateToday());
}
```

Я выполнил тестирование, а затем вновь изменил методы доступа, чтобы они работали с контрактом.

```
class Customer...
get discountRate() {return this._contract.discountRate;}
_setDiscountRate(aNumber) {this._contract.discountRate = aNumber;}
```

Поскольку я использую JavaScript, объявленного исходного поля нет, так что мне не нужно ничего удалять.

Изменение чистой записи

Этот рефакторинг обычно проще выполняется с объектами, поскольку инкапсуляция обеспечивает естественный способ "заворачивания" доступа к данным в методы. Если у меня есть много функций, обращающихся к чистой записи, то, хотя этот рефакторинг остается весьма ценным, он становится гораздо сложнее.

Я могу создавать функции доступа и изменять все операции чтения и записи, чтобы использовать их. Если перемещаемое поле является неизменяемым, то при установке значения я могу обновить как исходное, так и целевое поля, и постепенно перенести его чтения. Тем не менее моим первым шагом по возможности будет использование рефакторинга *Инкапсуляция записи* (с. 208), чтобы превратить запись в класс, что позволит легче вносить все необходимые изменения.

Пример: перенос в совместно используемый объект

Теперь давайте рассмотрим другой случай. Вот счет с процентной ставкой.

```
class Account...
constructor(number, type, interestRate) {
  this._number = number;
  this._type = type;
  this._interestRate = interestRate;
}
get interestRate() {return this._interestRate;}

class AccountType...
constructor(nameString) {
  this._name = nameString;
}
```

Я хочу изменить программу так, чтобы процентная ставка определялась исходя из типа счета.

Доступ к процентной ставке уже инкапсулирован, поэтому я просто создам поле и соответствующий метод доступа для типа счета.

```
class AccountType...
constructor(nameString, interestRate) {
  this._name = nameString;
  this._interestRate = interestRate;
}
get interestRate() {return this._interestRate;}
```

При обновлении методов доступа из счета есть потенциальная проблема. До этого рефакторинга у каждого счета была своя процентная ставка. Теперь я хочу, чтобы все счета совместно использовали процентные ставки соответствующего типа счета. Если все счета одного типа имеют одинаковую процентную ставку, наблюдаемое поведение не изменится, так что с этим рефакторингом не будет

никаких проблем. Но если в наличии есть счет с другой процентной ставкой, то мои действия перестают быть рефакторингом. Если данные счета хранятся в базе данных, я должен проверить ее и убедиться, что процентная ставка всех счетов соответствует их типам. Я могу также применить рефакторинг *Введение утверждения* (с. 346) для класса счета.

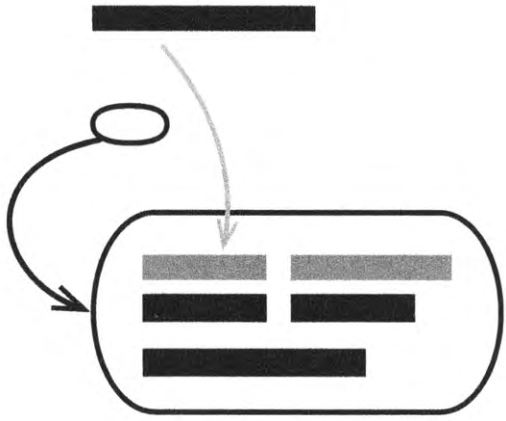
```
class Account...
constructor(number, type, interestRate) {
  this._number = number;
  this._type = type;
  assert(interestRate === this._type.interestRate);
  this._interestRate = interestRate;
}
get interestRate() {return this._interestRate;}
```

Я мог бы оставить систему на некоторое время с этим утверждением, чтобы увидеть, не обнаружится ли ошибка. Вместо добавления утверждения можно протоколировать проблему в журнале. И только убедившись, что я не вношу видимых изменений, я могу изменить доступ, полностью удалив обновление из счета.

```
class Account...
constructor(number, type) {
  this._number = number;
  this._type = type;
}
get interestRate() {return this._type.interestRate;}
```

Перенос инструкций в функцию (Move Statements into Function)

Обратный к рефакторингу *Перенос инструкций в точку вызова* (с. 262).



```
result.push(`<p>title: ${person.photo.title}</p>`);
result.concat(photoData(person.photo));

function photoData(aPhoto) {
  return [
    `<p>location: ${aPhoto.location}</p>`,
    `<p>date: ${aPhoto.date.toDateString()}</p>`,
  ];
}
```

```
result.concat(photoData(person.photo));

function photoData(aPhoto) {
  return [
    `<p>title: ${aPhoto.title}</p>`,
    `<p>location: ${aPhoto.location}</p>`,
    `<p>date: ${aPhoto.date.toDateString()}</p>`,
  ];
}
```

Мотивация

Удаление дублирования является одним из лучших практических правил для получения хорошего кода. Если один и тот же код выполняется каждый раз, когда вы вызываете определенную функцию, стоит подумать о переносе этого повторяющегося кода в саму функцию. Таким образом, любые будущие изменения повторяющегося кода могут выполняться в одном месте и использоваться во всех вызовах. Если в будущем этот код изменится, можно легко переместить его (или его часть) снова с помощью рефакторинга *Перенос инструкций в точку вызова* (с. 262).

Я переношу инструкции в функцию, когда понять эти инструкции легче как часть вызываемой функции. Если они не имеют смысла в виде части вызываемой функции, но все равно должны вызываться вместе с ней, я просто пользуюсь рефакторингом *Извлечение функции* (с. 152) для этих инструкций и вызываемой функции. По сути, это тот же процесс, который я опишу ниже, но без шагов встраивания и переименования. Нет ничего необычного в том, чтобы сделать все, кроме этих шагов, которые можно выполнить позже, после тщательных размышлений.

Техника

- Если повторяющийся код не соседствует с вызовом целевой функции, используйте рефакторинг *Перемещение инструкций* (с. 269), чтобы поместить его рядом с функцией.

- Если целевая функция вызывается только в одном месте, просто удалите код из его исходного местоположения, вставьте в целевую функцию, выполните тестирование и проигнорируйте остальную часть этой техники.
- Если у вас есть несколько мест вызова, примените рефакторинг *Извлечение функции* (с. 152) к одному из них, чтобы извлечь как вызов целевой функции, так и инструкции, которые вы хотите переместить в нее. Дайте ей временное имя, которое будет легко найти.
- Преобразуйте все остальные вызовы таким образом, чтобы в них использовалась новая функция. Выполняйте тестирование после каждого преобразования.
- Когда все исходные вызовы будут использовать новую функцию, примените рефакторинг *Встраивание функции* (с. 161), чтобы полностью встроить исходную функцию в новую, удалив при этом исходную функцию.
- Выполните рефакторинг *Переименование функции* (с. 170), чтобы изменить имя новой функции, дав ей имя исходной функции.

  Или лучшее имя, если таковое имеется.

Пример

Я начну со следующего кода, который выдает HTML-текст для данных о фотографии.

```
function renderPerson(outStream, person) {
  const result = [];
  result.push(`<p>${person.name}</p>`);
  result.push(renderPhoto(person.photo));
  result.push(`<p>title: ${person.photo.title}</p>`);
  result.push(emitPhotoData(person.photo));
  return result.join("\n");
}

function photoDiv(p) {
  return [
    "<div>",
    `<p>title: ${p.title}</p>`,
    emitPhotoData(p),
    "</div>",
  ].join("\n");
}

function emitPhotoData(aPhoto) {
  const result = [];
  result.push(`<p>location: ${aPhoto.location}</p>`);
  result.push(`<p>date: ${aPhoto.date.toDateString()}</p>`);
  return result.join("\n");
}
```

В этом коде показаны два вызова `emitPhotoData`, каждому из которых предшествует строка кода; семантически эти строки эквивалентны. Я хотел бы удалить дублирование, переместив вывод заголовка в `emitPhotoData`. Если бы у меня было только одно место вызова, я бы просто вырезал и перенес код, но чем больше у меня мест вызова, тем сильнее я склоняюсь к использованию более безопасной процедуры.

Начну с применения рефакторинга *Извлечение функции* (с. 152) к одному из мест вызова. Я извлекаю инструкции, которые хочу переместить в `emitPhotoData`, вместе с вызовом самой `emitPhotoData`.

```
function photoDiv(p) {
  return [
    "<div>",
    zznew(p),
    "</div>",
  ].join("\n");
}

function zznew(p) {
  return [
    `<p>title: ${p.title}</p>`,
    emitPhotoData(p),
  ].join("\n");
}
```

Теперь можно посмотреть на другие вызовы `emitPhotoData` и поочередно заменить вызовы и предшествующие им инструкции вызовами новой функции.

```
function renderPerson(outStream, person) {
  const result = [];
  result.push(`<p>${person.name}</p>`);
  result.push(renderPhoto(person.photo));
  result.push(zznew(person.photo));
  return result.join("\n");
}
```

Завершив работу с вызовами, я применяю рефакторинг *Встраивание функции* (с. 161) к `emitPhotoData`:

```
function zznew(p) {
  return [
    `<p>title: ${p.title}</p>`,
    `<p>location: ${p.location}</p>`,
    `<p>date: ${p.date.toDateString()}</p>`,
  ].join("\n");
}
```

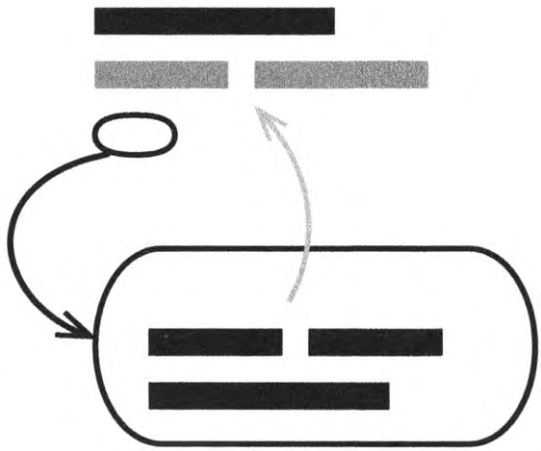
и завершаю работу применением рефакторинга *Переименование функции* (с. 170):

```
function renderPerson(outStream, person) {
  const result = [];
  result.push(`<p>${person.name}</p>`);
  result.push(renderPhoto(person.photo));
  result.push(emitPhotoData(person.photo));
  return result.join("\n");
}

function photoDiv(aPhoto) {
  return [
    "<div>",
    emitPhotoData(aPhoto),
    "</div>",
  ].join("\n");
}

function emitPhotoData(aPhoto) {
  return [
    `<p>title: ${aPhoto.title}</p>`,
    `<p>location: ${aPhoto.location}</p>`,
    `<p>date: ${aPhoto.date.toDateString()}</p>`,
  ].join("\n");
}
```

Кроме того, во время работы я заодно делаю имена параметров соответствующими моим соглашениям.

Перенос инструкций в точку вызова (Move Statements to Callers)

Обратный к рефакторингу *Перенос инструкций в функцию (с. 258)*.



```
emitPhotoData(outStream, person.photo);

function emitPhotoData(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>location: ${photo.location}</p>\n`);
}
```

```
emitPhotoData(outStream, person.photo);
outStream.write(`<p>location: ${person.photo.location}</p>\n`);

function emitPhotoData(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
}
```

Мотивация

Функции являются основными строительными блоками абстракций, которые мы строим как программисты. И, как и в случае любой абстракции, мы не всегда правильно определяем ее границы. По мере того как возможности базы кода изменяются (что естественно для полезного развивающегося программного обеспечения), часто обнаруживается, что меняются и границы абстракций. Для функций это означает, что то, что когда-то было сплоченной, атомарной единицей поведения, может стать смесью двух или более разных поведений.

Одним из спусковых крючков для этого является случай, когда общее поведение, используемое в нескольких местах, в некоторых вызовах должно меняться. В таком случае нужно переместить изменяющееся поведение из функции в вызывающую ее функцию. Я воспользуюсь рефакторингом *Перемещение инструкций* (с. 269), чтобы перенести изменяющееся поведение в начало или конец функции, а затем перемещу соответствующие инструкции в вызывающие функции. Как только изменяющийся код окажется в вызывающей функции, я смогу при необходимости его изменять.

Перемещение операторов в вызывающие абоненты хорошо работает для небольших изменений, но иногда границы между вызывающим и вызываемым абонентами нуждаются в полной переработке. В этом случае мой лучший ход — использовать встроенную функцию (*Встраивание функции* (с. 161)), а затем сдвигать и извлекать новые функции, чтобы сформировать лучшие границы.

Данный рефакторинг хорошо работает для небольших изменений, но иногда границы между вызывающим и вызываемым кодом нуждаются в полной переработке. В этом случае мой выбор — использовать рефакторинг *Встраивание функции* (с. 161), а затем перемещать код и извлекать новые функции, чтобы сформировать более соответствующие ситуации границы.

Техника

- В простых ситуациях, когда у вас есть только один или два вызова и простая вызываемая функция, — просто вырежьте первую строку из вызываемой функции и вставьте ее (возможно, с соответствующей подгонкой) в место вызова. Выполните тестирование и на этом завершите рефакторинг.
- В противном случае примените рефакторинг *Извлечение функции* (с. 152) ко всем инструкциям, которые вы *не* хотите перемещать; дайте функции временное имя, которое легко искать.

  Если функция является методом, перекрываемым в подклассах, выполните извлечение для всех подклассов, чтобы оставшийся метод был одинаковым во всех классах. Затем удалите методы подклассов.
- Примените к исходной функции рефакторинг *Встраивание функции* (с. 161).
- Примените к извлеченной функции рефакторинг *Изменение объявления функции* (с. 170), чтобы вернуть ей ее исходное имя.

  Или дайте ей лучшее имя, если у вас на примете есть таковое.

Пример

Вот простой пример — дважды вызываемая функция.

```
function renderPerson(outStream, person) {
  outStream.write(`<p>${person.name}</p>\n`);
  renderPhoto(outStream, person.photo);
  emitPhotoData(outStream, person.photo);
}

function listRecentPhotos(outStream, photos) {
  photos
    .filter(p => p.date > recentDateCutoff())
    .forEach(p => {
        outStream.write("<div>\n");
        emitPhotoData(outStream, p);
        outStream.write("</div>\n");
  });
}

function emitPhotoData(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>date: ${photo.date.toDateString()}</p>\n`);
  outStream.write(`<p>location: ${photo.location}</p>\n`);
}
```

Требуется изменить программу таким образом, чтобы функция `listRecentPhotos` отображала информацию о местоположении иначе, в то время как функция `renderPerson` осталась прежним. Чтобы упростить такое изменение, применю рассматриваемый рефакторинг к последней строке.

Обычно, сталкиваясь с чем-то подобным, я просто вырезаю последнюю строку из `renderPerson` и вставляю ее ниже двух вызовов функции. Но, поскольку я объясняю, что делать в более сложных случаях, я проведу вас по более сложному, но и более безопасному пути.

Первый шаг — использовать рефакторинг *Извлечение функции* (с. 152) для кода, который останется в функции `emitPhotoData`.

```
function renderPerson(outStream, person) {
  outStream.write(`<p>${person.name}</p>\n`);
  renderPhoto(outStream, person.photo);
  emitPhotoData(outStream, person.photo);
}

function listRecentPhotos(outStream, photos) {
  photos
    .filter(p => p.date > recentDateCutoff())
    .forEach(p => {
        outStream.write("<div>\n");
        emitPhotoData(outStream, p);
        outStream.write("</div>\n");
  });
}

function emitPhotoData(outStream, photo) {
  zztmp(outStream, photo);
  outStream.write(`<p>location: ${photo.location}</p>\n`);
}

function zztmp(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>date: ${photo.date.toDateString()}</p>\n`);
}
```

Обычно имя извлеченной функции носит временный характер, поэтому я не беспокоюсь о том, чтобы придумать что-нибудь значимое. Тем не менее для временного имени полезно использовать что-то, что легко найти и не спутать с чем-то иным. На этом этапе можно выполнить тестирование, чтобы убедиться, что код работает за границей вызова функции.

Теперь я использую рефакторинг *Встраивание функции* (с. 161) по одному вызову за раз. Начну с `renderPerson`.

```
function renderPerson(outStream, person) {
  outStream.write(`<p>${person.name}</p>\n`);
```

```
  renderPhoto(outStream, person.photo);
  zztmp(outStream, person.photo);
  outStream.write(`<p>location: ${person.photo.location}</p>\n`);
}

function listRecentPhotos(outStream, photos) {
  photos
    .filter(p => p.date > recentDateCutoff())
    .forEach(p => {
        outStream.write("<div>\n");
        emitPhotoData(outStream, p);
        outStream.write("</div>\n");
  });
}

function emitPhotoData(outStream, photo) {
  zztmp(outStream, photo);
  outStream.write(`<p>location: ${photo.location}</p>\n`);
}

function zztmp(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>date: ${photo.date.toDateString()}</p>\n`);
}
```

Я вновь прибегаю к тестированию, чтобы убедиться в корректной работе этого вызова, а затем выполняю следующий перенос.

```
function renderPerson(outStream, person) {
  outStream.write(`<p>${person.name}</p>\n`);
  renderPhoto(outStream, person.photo);
  zztmp(outStream, person.photo);
  outStream.write(`<p>location: ${person.photo.location}</p>\n`);
}

function listRecentPhotos(outStream, photos) {
  photos
    .filter(p => p.date > recentDateCutoff())
    .forEach(p => {
        outStream.write("<div>\n");
        zztmp(outStream, p);
        outStream.write(`<p>location: ${p.location}</p>\n`);
        outStream.write("</div>\n");
  });
}

function emitPhotoData(outStream, photo) {
  zztmp(outStream, photo);
  outStream.write(`<p>location: ${photo.location}</p>\n`);
}
```

```
function zztmp(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>date: ${photo.date.toDateString()}</p>\n`);
}
```

Теперь можно удалить внешнюю функцию, завершая рефакторинг *Встраивание функции* (с. 161).

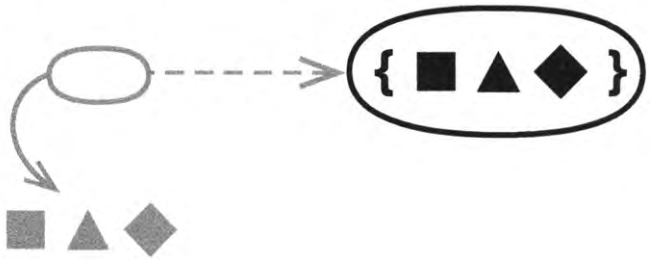
```
function renderPerson(outStream, person) {
  outStream.write(`<p>${person.name}</p>\n`);
  renderPhoto(outStream, person.photo);
  zztmp(outStream, person.photo);
  outStream.write(`<p>location: ${person.photo.location}</p>\n`);
}

function listRecentPhotos(outStream, photos) {
  photos
    .filter(p => p.date > recentDateCutoff())
    .forEach(p => {
        outStream.write("<div>\n");
        zztmp(outStream, p);
        outStream.write(`<p>location: ${p.location}</p>\n`);
        outStream.write("</div>\n");
  });
}

~~function emitPhotoData(outStream, photo) {~~
~~  zztmp(outStream, photo);~~
~~  outStream.write(`<p>location: ${photo.location}</p>\n`);~~
~~}~~

function zztmp(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>date: ${photo.date.toDateString()}</p>\n`);
}
```

Далее я переименовываю `zztmp`, возвращая ей исходное имя.

```
function renderPerson(outStream, person) {
  outStream.write(`<p>${person.name}</p>\n`);
  renderPhoto(outStream, person.photo);
  emitPhotoData(outStream, person.photo);
  outStream.write(`<p>location: ${person.photo.location}</p>\n`);
}

function listRecentPhotos(outStream, photos) {
  photos
    .filter(p => p.date > recentDateCutoff())
    .forEach(p => {
        outStream.write("<div>\n");
        emitPhotoData(outStream, p);
```

```
        outStream.write(`<p>location: ${p.location}</p>\n`);
        outStream.write("</div>\n");
  });
}

function emitPhotoData(outStream, photo) {
  outStream.write(`<p>title: ${photo.title}</p>\n`);
  outStream.write(`<p>date: ${photo.date.toDateString()}</p>\n`);
}
```

Замена встроенного кода вызовом функции (Replace Inline Code with Function Call)



```
let appliesToMass = false;
for(const s of states) {
  if (s === "MA") appliesToMass = true;
}
```

```
appliesToMass = states.includes("MA");
```

Мотивация

Функции позволяют упаковывать фрагменты поведения. Это полезно для понимания — именованная функция может объяснить назначение кода, а не его механику. Важно также удалить дублирование: вместо того, чтобы писать один и тот же код дважды, я просто вызываю функцию. Затем, чтобы изменить реализацию функции, мне не нужно отслеживать похожий код для внесения всех изменений. (Возможно, мне придется посмотреть на вызывающие функции, чтобы понять, все ли они должны использовать новый код, но это делается не так часто и выполняется намного проще.)

Встречая встроенный код, который делает то же самое, что и существующая функция, обычно я стараюсь заменить этот код вызовом функции. Исключение

составляют ситуации, когда я считаю, что сходство является случайным, так что если я изменю тело функции, то поведение этого встроенного кода не должно измениться. Руководством в такой ситуации является название функции. Если имя не имеет смысла, это может означать как то, что это плохое имя (в таком случае я использую рефакторинг *Переименование функции* (с. 170), чтобы исправить его), так и то, что назначение функции отличается от нужного мне в данной ситуации, и я не должен ее вызывать.

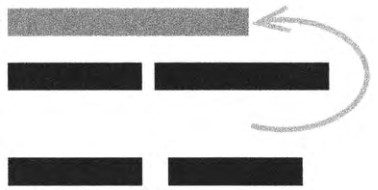
Особенно приятно работать с вызовами библиотечных функций — в этом случае мне даже не нужно писать тело функции.

Техника

- Замените встроенный код вызовом существующей функции.
- Выполните тестирование.

Перемещение инструкций (Slide Statements)

Бывший рефакторинг *Консолидация дублирующихся условных фрагментов*



```
const pricingPlan = retrievePricingPlan();
const order = retreiveOrder();
let charge;
const chargePerUnit = pricingPlan.unit;
```

```
const pricingPlan = retrievePricingPlan();
const chargePerUnit = pricingPlan.unit;
const order = retreiveOrder();
let charge;
```

Мотивация

Код легче понять, когда связанные друг с другом фрагменты находятся рядом. Если несколько строк кода обращаются к одной и той же структуре данных, то

лучше, чтобы они были вместе, а не были перемешаны с кодом, обращающимся к другим структурам данных. В простейшем случае я использую *Перемещение инструкций*, чтобы такой код представлял собой единое целое. Очень распространенным случаем является объявление и использование переменных. Некоторые программисты любят объявлять все свои переменные в верхней части функции; я же предпочитаю объявлять переменную непосредственно перед ее первым использованием.

Обычно я собираю связанный код вместе в качестве подготовительного шага для другого рефакторинга, которым часто оказывается рефакторинг *Извлечение функции* (с. 152). Помещение взаимосвязанного кода в четко выделенную функцию — это лучшее разделение, чем просто перемещение набора строк, чтобы они находились рядом. Но я не смогу выполнить рефакторинг *Извлечение функции* (с. 152), если извлекаемый код не будет собран вместе.

Техника

- Определите целевое местоположение для перемещения фрагмента кода. Изучите инструкции между исходным и целевым местоположениями, чтобы увидеть, нет ли каких-то помех для кандидата на перемещение. Если таковые имеются — откажитесь от дальнейших действий.

  Фрагмент не может быть перемещен назад далее объявления любого из элементов, к которым он обращается.

  Фрагмент не может быть перемещен вперед далее области видимости любого из элементов, к которым он обращается.

  Фрагмент не может быть перемещен через инструкцию, которая модифицирует любой из элементов, к которым он обращается.

  Фрагмент, изменяющий элемент, не может перемещаться через инструкцию, которая обращается к измененному элементу.

- Вырежьте фрагмент из исходного местоположения и вставьте его в целевую позицию.
- Выполните тестирование.

Если тест не пройден, попробуйте разбить перемещение на более мелкие шаги. Либо перемещайте фрагмент на меньшее расстояние, либо уменьшите объем перемещаемого фрагмента кода.

Пример

При перемещении фрагментов кода необходимо принять два решения: что именно я хочу переместить и куда я могу его переместить. Первое решение очень сильно зависит от контекста. На простейшем уровне мне нравится объявлять элементы рядом с тем местом, где я их использую, поэтому я часто перемещаю объявление элемента поближе к его использованию. Но почти всегда перемещение некоторого кода у меня связано с желанием выполнить другой рефакторинг — возможно, чтобы собрать некоторый код вместе для применения рефакторинга *Извлечение функции* (с. 152).

Как только у меня появится представление о том, куда именно я хочу переместить некоторый код, я должен решить, могу ли я это сделать. Принятие такого решения включает в себя просмотр кода, который я перемещаю, и кода, через который выполняется перемещение: не влияют ли они один на другой таким образом, что это меняет наблюдаемое поведение программы?

Рассмотрим следующий фрагмент кода.

```
 1 const pricingPlan = retrievePricingPlan();
 2 const order = retreiveOrder();
 3 const baseCharge = pricingPlan.base;
 4 let charge;
 5 const chargePerUnit = pricingPlan.unit;
 6 const units = order.units;
 7 let discount;
 8 charge = baseCharge + units * chargePerUnit;
 9 let discountableUnits =
     ↳ Math.max(units-pricingPlan.discountThreshold,0);
10 discount = discountableUnits * pricingPlan.discountFactor;
11 if (order.isRepeat) discount += 20;
12 charge = charge - discount;
13 chargeOrder(charge);
```

Первые семь строк являются объявлениями, и их относительно легко переместить. Например, я могу захотеть собрать вместе весь код, относящийся к скидкам, что означает перемещение строки 7 (`let discount`) в позицию над строкой 10 (`` `discount=...` ``). Поскольку объявление не имеет побочных эффектов и не обращается ни к какой другой переменной, я могу смело переместить объявление вперед до первой строки, которая обращается к скидке. Это распространенный шаг — если я хочу применить рефакторинг *Извлечение функции* (с. 152) к логике скидок, мне нужно сначала переместить объявление вниз.

Я провожу аналогичный анализ для любого кода, который не имеет побочных действий. Поэтому я без проблем могу взять строку 2 (`` `const order=...` ``) и переместить ее вниз до строки 6 (`` `const units=...` ``).

В этом случае мне помогает тот факт, что код, по которому я перемещаюсь, не имеет побочных действий. Такой код можно смело перемещать, что является одной из причин, почему мудрые программисты, насколько это возможно, предпочитают использовать код без побочных действий.

Однако здесь есть свои тонкости. Откуда мне знать, что строка 2 не имеет побочных действий? Чтобы быть в этом уверенным, мне нужно заглянуть внутрь функции `retrieveOrder()` и убедиться, что в ней нет побочных действий (как и внутри любых функций, которые она вызывает, и внутри любых функций, которые вызываются вызванными ею функциями, и так далее). На практике, работая над своим собственным кодом, я знаю, что обычно следую *принципу разделения команд и запросов* [20], поэтому любая функция, которая возвращает значение, не имеет побочных действий. Но я могу быть уверен в этом только потому, что знаю свою кодовую базу; при работе в неизвестной кодовой базе я должен быть более осторожным. В своем собственном коде я стараюсь строго следовать принципу разделения команд и запросов, потому что это очень важно — знать, что код не имеет побочных действий.

При перемещении кода, который имеет побочные действия (или при перемещении через код с побочными действиями), я должен быть намного осторожнее. Я ищу препятствия переносу между двумя фрагментами кода. Допустим, я хочу сдвинуть строку 11 (`` `if(order.isRepeat)...` ``) в конец кода. Я не могу сделать это, потому что в строке 12 имеется обращение к переменной, состояние которой я изменяю в строке 11. Точно так же я не могу взять строку 13 (`` `chargeOrder(charge)` ``) и переместить ее вверх, потому что строка 12 изменяет некоторое состояние, к которому обращается строка 13. Тем не менее я могу переместить строку 8 (`` `charge=baseCharge+...` ``) через строки 9–11, потому что они не изменяют никакое общее состояние.

Самое простое правило, которому нужно следовать, состоит в том, что я не могу перемещать один фрагмент кода через другой, если какие-либо данные, к которым обращаются оба фрагмента, одним из них изменяются. Но это не всеобъемлющее правило — я могу поменять местами следующие две строки:

```
a = a + 10;
a = a + 5;
```

Но обоснование безопасности перемещения требует от меня действительного понимания, какие операции выполняются и как они скомпонованы.

Поскольку я вынужден пристально следить за обновлением состояния, я стараюсь максимально его сократить. Я применяю рефакторинг *Расщепление переменной* (с. 285) к `charge`, прежде чем позволить себе перенос в этом коде.

В данном случае анализ относительно прост, потому что я в основном просто модифицирую локальные переменные. С более сложными структурами данных

гораздо труднее быть уверенным в отсутствии проблем. Поэтому тесты играют особенно важную роль: переместив малейший фрагмент, выполните тесты и посмотрите, все ли работает как надо. Если мой охват тестами достаточен, я могу чувствовать себя спокойным. Но если тесты ненадежны, нужно быть предельно осторожным или, что еще лучше, — усовершенствовать тесты для кода, с которым идет работа.

Наиболее важным следствием непройденного теста после перемещения является использование меньших перемещений: вместо того, чтобы перемещать десять строк, я попытаюсь переместить пять, или уменьшу величину перемещения. Сбойные тесты могут также означать, что перемещение того не стоит, и мне нужно сначала поработать над чем-то другим.

Пример: перемещение с условными конструкциями

Я могу выполнять перемещения и с условными конструкциями. Это либо приводит к удалению дублирования логики, когда я выполняю перемещение из условной конструкции, либо к добавлению при перемещении внутрь нее.

Вот ситуация, когда у меня есть одинаковые инструкции в обеих ветвях условной конструкции:

```
let result;
if (availableResources.length === 0) {
  result = createResource();
  allocatedResources.push(result);
} else {
  result = availableResources.pop();
  allocatedResources.push(result);
}
return result;
```

Я могу вынести их из условной конструкции, и в этом случае они превратятся в единственную конструкцию вне блока условной конструкции.

```
let result;
if (availableResources.length === 0) {
  result = createResource();
} else {
  result = availableResources.pop();
}
allocatedResources.push(result);
return result;
```

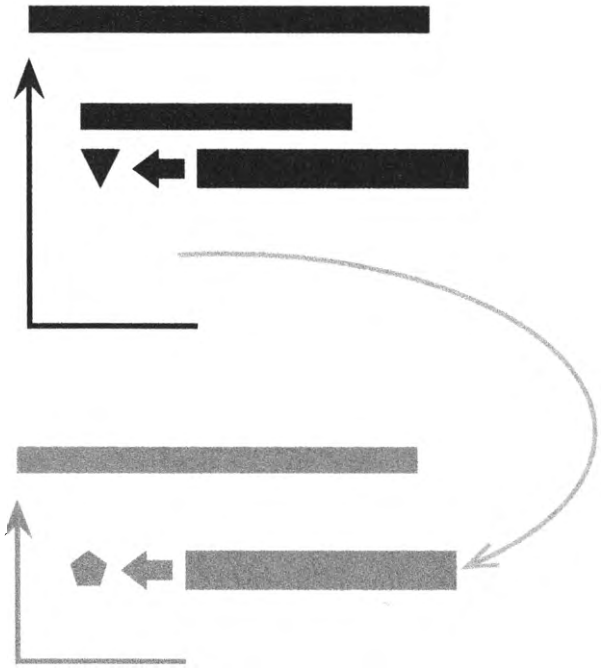
В обратном случае перемещение фрагмента внутрь условной конструкции означает его повторение в каждой ветви.

Дальнейшее чтение

Я видел почти идентичный рефакторинг под названием *Обмен инструкций* [48]. Он перемещает соседние фрагменты, но работает только с фрагментами размером в одну инструкцию. Вы можете думать о нем, как о рефакторинге *Перемещение инструкций*, где и перемещаемый фрагмент, и фрагмент, через который выполняется перемещение, представляют собой отдельные инструкции. Этот рефакторинг выглядит привлекательно; в конце концов, я всегда стараюсь делать маленькие шаги — шаги, которые могут показаться новичкам в рефакторинге смехотворно маленькими.

Но в итоге я написал этот рефакторинг с большими фрагментами. Я стараюсь перемещать инструкции по одной, только если у меня возникают трудности с большими фрагментами, но я редко сталкиваюсь с проблемами, при решении которых требуется перемещение фрагментов большого размера. Однако в случае плохого кода работа с фрагментами меньшего размера может оказаться существенно более легкой.

Разделение цикла (Split Loop)



```
let averageAge = 0;
let totalSalary = 0;
for (const p of people) {
  averageAge += p.age;
  totalSalary += p.salary;
}
averageAge = averageAge / people.length;
```

```
let totalSalary = 0;
for (const p of people) {
  totalSalary += p.salary;
}

let averageAge = 0;
for (const p of people) {
  averageAge += p.age;
}
averageAge = averageAge / people.length;
```

Мотивация

Часто приходится встречать циклы, которые решают одновременно разные задачи только потому, что они могут сделать это за один проход цикла. Но если вы выполняете в одном и том же цикле разные действия, то всякий раз, когда вам нужно изменить цикл, вы должны понимать все решаемые им задачи. Разбивая цикл, вы гарантируете, что вам нужно понимать только то поведение, которое требуется изменить.

Разделение цикла также может облегчить его использование. Цикл, который вычисляет одно значение, может просто его вернуть. Циклы, которые решают много задач, должны возвращать структуры или заполнять локальные переменные. Я часто использую последовательность рефакторингов, в которой за рефакторингом *Разделение цикла* следует рефакторинг *Извлечение функции* (с. 152).

Многим программистам такой рефакторинг не нравится, так как он заставляет выполнять цикл многократно. Мое напоминание, как обычно, заключается в том, что рефакторинг и оптимизация — несколько разные вещи. Как только мой код станет ясным, я оптимизирую его, и, если работа цикла окажется узким местом, будет легко соединить циклы в один. Но на самом деле итерирование даже большого списка редко является узким местом, а разделение циклов часто дает возможность других, более мощных оптимизаций.

Техника

- Скопируйте цикл.
- Определите и устраните дублирующиеся побочные действия.
- Выполните тестирование.

По окончании работы подумайте о применении рефакторинга *Извлечение функции* (с. 152) к каждому циклу.

Пример

Начну с небольшого кода, который рассчитывает общую зарплату и младший возраст группы людей.

```
let youngest = people[0] ? people[0].age : Infinity;
let totalSalary = 0;
for (const p of people) {
  if (p.age < youngest) youngest = p.age;
  totalSalary += p.salary;
}
return `youngestAge: ${youngest}, totalSalary: ${totalSalary}`;
```

Это очень простой цикл, но он выполняет два разных вычисления. Чтобы разделить их, я начинаю с простого копирования цикла.

```
let youngest = people[0] ? people[0].age : Infinity;
let totalSalary = 0;
for (const p of people) {
  if (p.age < youngest) youngest = p.age;
  totalSalary += p.salary;
}
for (const p of people) {
  if (p.age < youngest) youngest = p.age;
  totalSalary += p.salary;
}
return `youngestAge: ${youngest}, totalSalary: ${totalSalary}`;
```

При наличии скопированного цикла мне нужно удалить дублирование, которое в противном случае даст неправильные результаты. Если некоторый код в цикле не имеет побочных действий, я могу пока оставить его на месте, но в данном примере это не так.

```
let youngest = people[0] ? people[0].age : Infinity;
let totalSalary = 0;
for (const p of people) {
  if (p.age < youngest) youngest = p.age;
  totalSalary += p.salary;
}
```

```
for (const p of people) {
  if (p.age < youngest) youngest = p.age;
  totalSalary += p.salary;
}
return `youngestAge: ${youngest}, totalSalary: ${totalSalary}`;
```

Официально это конец рефакторинга *Разделение цикла*. Но его смысл не в том, что он делает сам по себе, а в том, что он выполняет подготовку к следующему ходу, а я обычно стремлюсь извлечь циклы в их собственные функции. Я воспользуюсь рефакторингом *Перемещение инструкций* (с. 269), чтобы немного реорганизовать код.

```
let totalSalary = 0;
for (const p of people) {
  totalSalary += p.salary;
}

let youngest = people[0] ? people[0].age : Infinity;
for (const p of people) {
  if (p.age < youngest) youngest = p.age;
}

return `youngestAge: ${youngest}, totalSalary: ${totalSalary}`;
```

Теперь я выполняю пару рефакторингов *Извлечение функции* (с. 152).

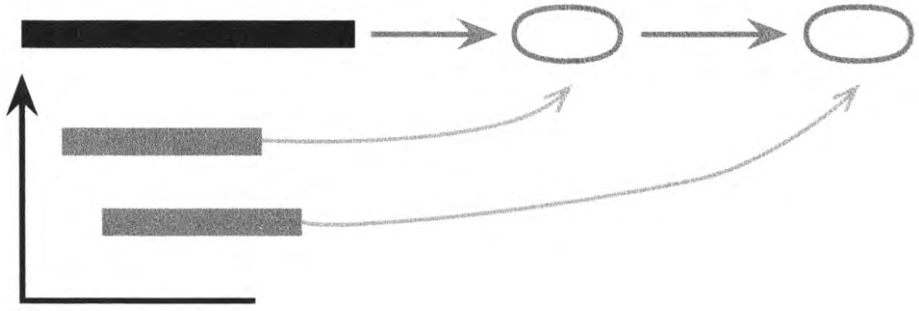
```
return `youngestAge: ${youngestAge()}, ` +
       `totalSalary: ${totalSalary()}`;

function totalSalary() {
  let totalSalary = 0;
  for (const p of people) {
    totalSalary += p.salary;
  }
  return totalSalary;
}

function youngestAge() {
  let youngest = people[0] ? people[0].age : Infinity;
  for (const p of people) {
    if (p.age < youngest) youngest = p.age;
  }
  return youngest;
}
```

Я не могу сопротивляться желанию выполнить рефакторинг *Замена цикла конвейером* (с. 278) для общей зарплаты, а для вычисления младшего возраста так и просится рефакторинг *Подстановка алгоритма* (с. 240).

```
return `youngestAge: ${youngestAge()}, ` +
       `totalSalary: ${totalSalary()}`;
```

```
function totalSalary() {
  return people.reduce((total,p) => total + p.salary, 0);
}
function youngestAge() {
  return Math.min(...people.map(p => p.age));
}
```

Замена цикла конвейером (Replace Loop with Pipeline)



```
const names = [];
for (const i of input) {
  if (i.job === "programmer")
    names.push(i.name);
}
```

```
const names = input
    .filter(i => i.job === "programmer")
    .map(i => i.name)
;
```

Мотивация

Как и большинство программистов, меня учили использовать циклы для обхода коллекции объектов. Однако все чаще языковые среды обеспечивают для этого лучшую конструкцию — конвейер коллекции. Конвейеры коллекций [19] позволяют мне описать обработку как последовательность операций, каждая из которых потребляет и генерирует коллекцию. Наиболее распространенными из этих операций являются *отображение* (map), использующее функции для преобразования каждого элемента входной коллекции, и *фильтр* (filter), в котором используется функция для выбора подмножества входной коллекции для последующих действий в конвейере. Мне гораздо проще следовать логике, если она выражена в

виде конвейера — в таком случае я могу читать ее сверху вниз, чтобы увидеть, как объекты проходят через конвейер.

Техника

- Создайте новую переменную для коллекции цикла.

  Это может быть простая копия существующей переменной.
- В нисходящем направлении возьмите каждый фрагмент поведения в цикле и замените его операцией конвейера коллекции в выводе переменной сбора цикла. Выполните тестирование после каждого изменения.
- Как только все поведение будет убрано из цикла, удалите его.

  Если цикл выполняет присваивание аккумулятору, присвойте аккумулятору результат работы конвейера.

Пример

Я начну с некоторых данных: CSV-файл данных о наших офисах.

```
office, country, telephone
Chicago, USA, +1 312 373 1000
Beijing, China, +86 4008 900 505
Bangalore, India, +91 80 4064 9570
Porto Alegre, Brazil, +55 51 3079 3550
Chennai, India, +91 44 660 44766
... (данные о других офисах)
```

Следующая функция выбирает офисы в Индии и возвращает их города и номера телефонов.

```
function acquireData(input) {
  const lines = input.split("\n");
  let firstLine = true;
  const result = [];
  for (const line of lines) {
    if (firstLine) {
      firstLine = false;
      continue;
    }
    if (line.trim() === "") continue;
    const record = line.split(",");
    if (record[1].trim() === "India") {
      result.push({city:record[0].trim(),phone:record[2].trim()});
    }
  }
  return result;
}
```

Я хочу заменить этот цикл конвейером коллекции.

Мой первый шаг состоит в создании отдельной переменной для цикла.

```
function acquireData(input) {
  const lines = input.split("\n");
  let firstLine = true;
  const result = [];
  const loopItems = lines
  for (const line of loopItems) {
    if (firstLine) {
      firstLine = false;
      continue;
    }
    if (line.trim() === "") continue;
    const record = line.split(",");
    if (record[1].trim() === "India") {
      result.push({city:record[0].trim(),phone:record[2].trim()});
    }
  }
  return result;
}
```

Первая часть цикла посвящена пропуску первой строки файла CSV. Это можно сделать с помощью операции `slice`, поэтому я удаляю первый раздел цикла и добавляю операцию среза к создаваемой переменной цикла.

```
function acquireData(input) {
  const lines = input.split("\n");
  let firstLine = true;
  const result = [];
  const loopItems = lines
        .slice(1);
  for (const line of loopItems) {
    if (firstLine) {
      firstLine = false;
      continue;
    }
    if (line.trim() === "") continue;
    const record = line.split(",");
    if (record[1].trim() === "India") {
      result.push({city:record[0].trim(),phone:record[2].trim()});
    }
  }
  return result;
}
```

В качестве бонуса это позволяет мне удалить `firstLine` — мне особенно нравится удалять управляющие переменные.

Следующий фрагмент поведения удаляет все пустые строки. Я могу заменить это операцией фильтра.

```
function acquireData(input) {
  const lines = input.split("\n");
  const result = [];
  const loopItems = lines
          .slice(1)
          .filter(line => line.trim() !== "")
          ;
  for (const line of loopItems) {
    if (line.trim() === "") continue;
    const record = line.split(",");
    if (record[1].trim() === "India") {
      result.push({city:record[0].trim(),phone:record[2].trim()});
    }
  }
  return result;
}
```

При написании конвейера я предпочитаю ставить завершающую точку с запятой на отдельной строке.

Я использую операцию map, чтобы превратить строки в массив строк — ошибочно названная записью (record) операция исходной функции, имя которой пока что безопаснее сохранить как есть и переименовать ее позже.

```
function acquireData(input) {
  const lines = input.split("\n");
  const result = [];
  const loopItems = lines
          .slice(1)
          .filter(line => line.trim() !== "")
          .map(line => line.split(","))
          ;
  for (const line of loopItems) {
    const record = line;.split(",");
    if (record[1].trim() === "India") {
      result.push({city:record[0].trim(),phone:record[2].trim()});
    }
  }
  return result;
}
```

Выполняю фильтрацию для поиска записей, относящихся к Индии.

```
function acquireData(input) {
  const lines = input.split("\n");
  const result = [];
  const loopItems = lines
          .slice(1)
          .filter(line => line.trim() !== "")
          .map(line => line.split(","))
          .filter(record => record[1].trim() === "India")
          ;
```

```
  for (const line of loopItems) {
    const record = line;
    if (record[1].trim() === "India") {
      result.push({city: record[0].trim(), phone: record[2].trim()});
    }
  }
  return result;
}
```

Выполняю отображение для формирования выходных данных.

```
  function acquireData(input) {
  const lines = input.split("\n");
  const result = [];
  const loopItems = lines
      .slice(1)
      .filter(line => line.trim() !== "")
      .map(line => line.split(","))
      .filter(record => record[1].trim() === "India")
      .map(record=>({city:record[0].trim(),phone:record[2].trim()}))
      ;
  for (const line of loopItems) {
    const record = line;
    result.push(line);
  }
  return result;
}
```

Теперь все, что делает цикл, это присваивает значения аккумулятору. Поэтому я могу удалить его и присвоить аккумулятору результат работы конвейера.

```
function acquireData(input) {
  const lines = input.split("\n");
  const result = lines
      .slice(1)
      .filter(line => line.trim() !== "")
      .map(line => line.split(","))
      .filter(record => record[1].trim() === "India")
      .map(record=>({city:record[0].trim(),phone:record[2].trim()}))
      ;
  for (const line of loopItems) {
    const record = line;
    result.push(line);
  }
  return result;
}
```

В этом и состоит рефакторинг. Но хотел бы сделать некоторые "завершающие мазки". Я встраиваю `result`, переименовываю некоторые лямбда-переменные и делаю внешний вид кода более похожим на таблицу.

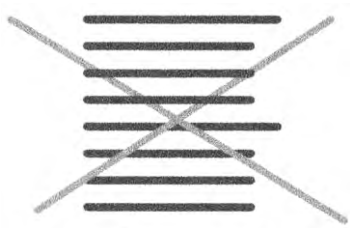
```
function acquireData(input) {
  const lines = input.split("\n");
  return lines
    .slice (1)
    .filter(line => line.trim() !== "")
    .map   (line => line.split(","))
    .filter(fields => fields[1].trim() === "India")
    .map   (fields=>({city:fields[0].trim(),phone:fields[2].trim()}))
    ;
}
```

Я подумывал также о встраивании `lines`, но почувствовал, что присутствие этой переменной поясняет происходящее.

Дальнейшее чтение

Дополнительные примеры превращения циклов в конвейеры вы можете найти в моем эссе *Рефакторинг с использованием циклов и конвейеров сбора* [32].

Удаление неработающего кода (Remove Dead Code)



```
if(false) {
  doSomethingThatUsedToMatter();
}
```

Мотивация

Когда мы выпускаем код "в свет", обычно нас мало волнует его размер. Несколько неиспользуемых строк кода не замедляют работу систем и не занимают значительную память; более того, современные умные компиляторы их просто удаляют. Но неиспользуемый код все еще является серьезным бременем при попытке понять, как работает программное обеспечение. Он не несет никаких

предупреждающих знаков, говорящих программистам, например, о том, что они могут игнорировать эту функцию, так как она больше не вызывается, и им приходится тратить время на понимание того, что она делает, и почему ее изменение не влияет на результат так, как они ожидали.

Если код больше не используется, его следует удалить. Я не волнуюсь, что он может понадобиться мне когда-нибудь в будущем; если это произойдет, у меня есть система контроля версий, и я всегда могу его восстановить. Если я уверен, что в один прекрасный день этот код действительно мне потребуется — я могу добавить в код комментарий, в котором упоминается этот удаленный код и дата его удаления (но, честно говоря, я не могу вспомнить, когда в последний раз это делал, или пожалел, что не сделал этого).

Распространенная привычка программистов — закомментировать мертвый код. Это было полезно в те времена, когда системы контроля версий еще широко не использовались или когда они были неудобны. Теперь, когда я могу поместить даже самую маленькую базу кода под управление такой системы, это больше не нужно.

Техника

- Если на мертвый код могут быть ссылки извне (например, когда это функция) — выполните поиск ее вызовов.
- Удалите мертвый код.
- Выполните тестирование.

Глава 9

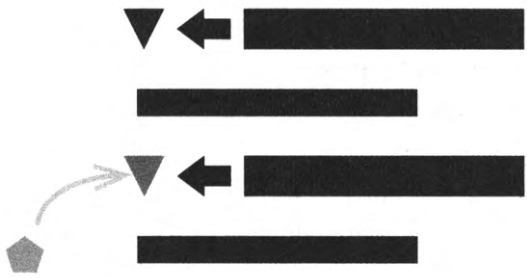
Организация данных

Структуры данных играют важную роль в наших программах, поэтому не удивительно, что у меня есть группа рефакторингов, которые фокусируются именно на них. Значение, которое используется для различных целей, является питательной средой для путаницы и ошибок, поэтому, обнаружив его, я использую рефакторинг *Расщепление переменной* (с. 285), чтобы разделить такие разноплановые использования. Как и в случае с любым другим элементом программы, правильно выбрать имя переменной — сложная и важная задача, поэтому я часто прибегаю к рефакторингу *Переименование переменной* (с. 183). Но иногда лучшее, что я могу сделать с переменной, — это полностью от нее избавиться с помощью рефакторинга *Замена вычисленной переменной запросом* (с. 293).

Я часто нахожу проблемы в базе кода из-за путаницы между ссылками и значениями и поэтому использую рефакторинги *Замена ссылки значением* (с. 296) и *Замена значения ссылкой* (с. 300) для переключения между этими стилями.

Расщепление переменной (Split Variable)

Бывший рефакторинг *Удаление присваиваний параметрам*
Бывший рефакторинг *Расщепление временной переменной*



```
let temp = 2 * (height + width);
console.log(temp);
temp = height * width;
console.log(temp);
```

```
const perimeter = 2 * (height + width);
console.log(perimeter);
const area = height * width;
console.log(area);
```

Мотивация

Переменные используются по-разному. Некоторые из их применений естественным образом приводят к неоднократным присваиваниям переменным. Например, переменные цикла меняются при каждой итерации цикла (например, `i` в `for(let i=0;i<10;i++)`). Переменные-аккумуляторы накапливают значение во время работы метода.

Многие другие переменные используются для хранения результата длинного фрагмента кода для последующего использования. Переменные этого вида присваиваются только один раз. Если они присваиваются более одного раза, это признак того, что в методе они имеют несколько ответственностей. Любая переменная с более чем одной ответственностью должна быть заменена несколькими переменными, по одной для каждой ответственности. Использование переменной для решения двух разных задач очень запутывает читателя.

Техника

- Измените имя переменной при ее объявлении и первом присваивании.

  Если последующие присваивания имеют вид `i=i+нечто`, то это аккумулирующая переменная; ее не следует расщеплять. Такие переменные часто используются для вычисления сумм, конкатенации строк, записи в поток или добавления в коллекцию.
- Если возможно, объявите новую переменную как неизменяемую.
- Измените все обращения к переменной до ее второго присваивания.
- Выполните тестирование.
- Повторите эти действия поэтапно, на каждом этапе переименовывая переменную в объявлении и меняя обращения к ней до следующего присваивания, пока не достигнете ее последнего присваивания.

Пример

В этом примере я вычисляю расстояние, пройденное объектом. С самого начала на объект воздействует некоторая первичная сила. После некоторой задержки включается вторичная сила, чтобы еще больше ускорить объект. Используя законы движения, я могу вычислить пройденное расстояние следующим образом.

```
function distanceTravelled (scenario, time) {
  let result;
  let acc = scenario.primaryForce / scenario.mass;
  let primaryTime = Math.min(time, scenario.delay);
  result = 0.5 * acc * primaryTime * primaryTime;
  let secondaryTime = time - scenario.delay;
  if (secondaryTime > 0) {
    let primaryVelocity = acc * scenario.delay;
    acc = (scenario.primaryForce + scenario.secondaryForce)
          /scenario.mass;
    result += primaryVelocity * secondaryTime
            + 0.5 * acc * secondaryTime * secondaryTime;
  }
  return result;
}
```

Хорошая маленькая запутанная функция. В примере для нас интересно то, что переменная `acc` присваивается дважды. У нее две ответственности: одна — для хранения начального ускорения от действия первой силы, а вторая — для хранения ускорения под воздействием обеих сил. Я хочу расщепить эту переменную.

При попытке понять, как используется переменная, удобно, если редактор может выделить все вхождения имени в функции или файле. Большинство современных редакторов довольно легко это делают.

Я начину с самого начала, меняя имя переменной и объявляя новое имя как `const`. Затем я изменяю все обращения к переменной от этой точки до следующего ее присваивания назначения. При следующем присваивании я объявляю ее:

```
function distanceTravelled (scenario, time) {
  let result;
  const primaryAcceleration = scenario.primaryForce/scenario.mass;
  let primaryTime = Math.min(time, scenario.delay);
  result = 0.5*primaryAcceleration*primaryTime*primaryTime;
  let secondaryTime = time - scenario.delay;
  if (secondaryTime > 0) {
    let primaryVelocity = primaryAcceleration*scenario.delay;
    let acc = (scenario.primaryForce + scenario.secondaryForce)
              /scenario.mass;
    result += primaryVelocity*secondaryTime
            + 0.5*acc*secondaryTime*secondaryTime;
  }
  return result;
}
```

Я выбираю новое имя только для первого использования переменной. Я объявляю ее как `const`, чтобы гарантировать, что она будет присвоена только один раз. Затем можно объявить исходную переменную во втором присваивании. Теперь я могу скомпилировать и протестировать код, и все должно работать.

Я продолжаю работу со вторым присваиванием переменной. В результате исходное имя переменной удаляется полностью, и переменная заменяется новой, названной специально для ее второго применения.

```
function distanceTravelled (scenario, time) {
  let result;
  const primaryAcceleration = scenario.primaryForce/scenario.mass;
  let primaryTime = Math.min(time, scenario.delay);
  result = 0.5*primaryAcceleration*primaryTime*primaryTime;
  let secondaryTime = time - scenario.delay;
  if (secondaryTime > 0) {
    let primaryVelocity = primaryAcceleration*scenario.delay;
    const secondaryAcceleration = (scenario.primaryForce
            +scenario.secondaryForce) / scenario.mass;
    result +=primaryVelocity * secondaryTime + 0.5
            *secondaryAcceleration*secondaryTime*secondaryTime;
  }
  return result;
}
```

Уверен, что вы можете придумать и другие рефакторинги для данного кода. Попробуйте и получите от них удовольствие!

Пример: присваивание входному параметру

Другой случай расщепления переменной — когда переменная объявлена как входной параметр. Рассмотрим такой код:

```
function discount (inputValue, quantity) {
  if (inputValue > 50) inputValue = inputValue - 2;
  if (quantity > 100) inputValue = inputValue - 1;
  return inputValue;
}
```

Здесь `inputValue` используется как для передачи входных данных в функцию, так и для хранения результата для возврата вызывающей функции. (Поскольку JavaScript передает параметры в функции по значению, любая модификация `inputValue` не влияет на вызывающую функцию.)

В этой ситуации предпочитаю расщепить эту переменную.

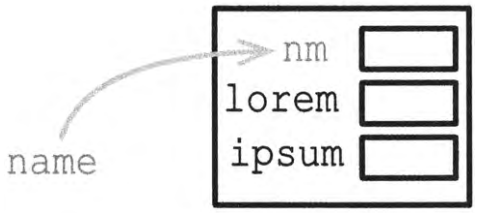
```
function discount (originalInputValue, quantity) {
  let inputValue = originalInputValue;
  if (inputValue > 50) inputValue = inputValue - 2;
  if (quantity > 100) inputValue = inputValue - 1;
  return inputValue;
}
```

Затем я дважды выполняю рефакторинг *Переименование переменной* (с. 183), чтобы дать переменным лучшие имена.

```
function discount (inputValue, quantity) {
  let result = inputValue;
  if (inputValue > 50 ) result = result - 2;
  if (quantity   > 100) result = result - 1;
  return result;
}
```

Обратите внимание: я изменил вторую строку, чтобы использовать в качестве источника данных именно `inputValue`. Хотя эти два значения одинаковы, я думаю, что на самом деле эта строка говорит о применении модификации к значению результата на основе исходного входного значения, а не (случайно такого же) значения аккумулятора результата.

Переименование поля (Rename Field)




```
class Organization {
  get name() {...}
}
```

```
class Organization {
  get title() {...}
}
```

Мотивация

Имена всегда важны, а имена полей в структурах записей могут быть особенно важны, когда эти структуры широко используются в программе. Структуры данных играют особенно важную роль в понимании. Много лет назад Фред Брукс (Fred Brooks) сказал: "Покажите мне свои блок-схемы и спрячьте таблицы — и я останусь в недоумении. Покажите мне свои таблицы, и мне не понадобятся ваши блок-схемы — они будут очевидны". Хотя ныне я не вижу, чтобы кто-то, кроме первокурсников, рисовал блок-схемы, сказанное остается в силе. Ключом к пониманию того, что происходит, являются структуры данных.

Поскольку структуры данных столь важны, нужно поддерживать их ясными и понятными. Как и все остальное, по мере работы с программой мое понимание данных улучшается, и очень важно, чтобы это улучшенное понимание было встроено в программу.

Возможно, вы захотите переименовать поле в структуре записи, но та же идея применима и к классам. Методы получения и установки значений формируют эффективное поле для пользователей класса. Их переименование так же важно, как и в случае простых структур записей.

Техника

- Если запись имеет ограниченную область видимости, переименуйте все обращения к полю и выполните тестирование; остальную часть техники делать не нужно.
- Если запись еще не инкапсулирована, примените рефакторинг *Инкапсуляция записи* (с. 208).
- Переименуйте закрытое поле внутри объекта и обновите соответствующим образом внутренние методы.
- Выполните тестирование.
- Если конструктор использует это имя, примените рефакторинг *Изменение объявления функции* (с. 170) для его переименования.
- Примените рефакторинг *Переименование функции* (с. 170) к методам доступа.

Пример: переименование поля

Начнем с константы.

```
const organization = {name: "Acme Gooseberries", country: "GB"};
```

Я хочу изменить "name" на "title". Этот объект широко используется в кодовой базе, и в коде есть обновления заголовка. Мой первый шаг — применение рефакторинга *Инкапсуляция записи* (с. 208).

```
class Organization {
  constructor(data) {
    this._name = data.name;
    this._country = data.country;
  }

  get name()         {return this._name;}
  set name(aString)  {this._name = aString;}
  get country()              {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}
```

```
const organization =
  new Organization({name: "Acme Gooseberries", country: "GB"});
```

Теперь, когда я инкапсулировал структуру записи в класс, есть четыре места, на которые мне нужно обратить внимание при переименовании: функция получения значения, функция установки значения, конструктор и внутренняя структура данных. Хотя это может выглядеть так, как будто я только увеличил количество работы, на самом деле моя работа стала легче, поскольку теперь я могу менять их независимо, а не все одновременно, предпринимая меньшие шаги. Меньшие шаги означают меньше вещей, которые могут пойти не так на каждом этапе — а следовательно, меньше неприятностей и меньше работы. Это не потребовалось бы, если бы я никогда не делал ошибок, но не делать ошибок — это миф, от которого я давно отказался.

Поскольку я скопировал структуру входных данных во внутреннюю структуру данных, мне нужно разделить их так, чтобы я мог работать с ними независимо. Я могу сделать это, определив отдельное поле и настроив конструктор и методы доступа для его использования.

```
class Organization...
class Organization {
  constructor(data) {
    this._title = data.name;
    this._country = data.country;
  }
  get name()         {return this._title;}
  set name(aString)  {this._title = aString;}
  get country()              {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}
```

Далее я добавляю поддержку использования "`title`" в конструкторе.

```
class Organization...
class Organization {
  constructor(data) {
    this._title =
      (data.title !== undefined) ? data.title : data.name;
    this._country = data.country;
  }
  get name()         {return this._title;}
  set name(aString)  {this._title = aString;}
  get country()              {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}
```

Теперь функции, вызывающие мой конструктор, могут использовать либо `name`, либо `title` (с приоритетом `title`). Следовательно, я могу просмотреть все вызовы конструктора и изменить их по одному, чтобы использовать новое имя.

```
const organization =
    new Organization({title: "Acme Gooseberries", country: "GB"});
```

Скорректировав их, я могу удалить поддержку name.

```
class Organization...
class Organization {
  constructor(data) {
    this._title = data.title;
    this._country = data.country;
  }
  get name()         {return this._title;}
  set name(aString)  {this._title = aString;}
  get country()              {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}
```

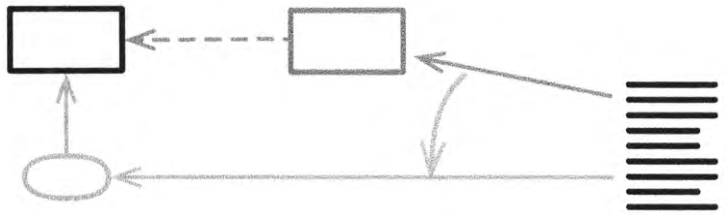
Теперь, когда конструктор и данные используют новое имя, можно изменить методы доступа, что так же просто, как и применение к каждому из них рефакторинга *Переименование функции* (с. 170).

```
class Organization...
class Organization {
  constructor(data) {
    this._title = data.title;
    this._country = data.country;
  }
  get title()         {return this._title;}
  set title(aString)  {this._title = aString;}
  get country()              {return this._country;}
  set country(aCountryCode) {this._country = aCountryCode;}
}
```

Я показал этот процесс в его наиболее тяжеловесной форме, необходимой для широко используемой структуры данных. Если она применяется только локально, например в единственной функции, — я, вероятно, смогу просто переименовать различные свойства за один раз без инкапсуляции. Это спорный вопрос — нужно ли применять всю описанную здесь технику, но, как обычно при применении рефакторинга, — если тесты не проходят, это признак того, что нужно использовать более многоступенчатую процедуру.

Некоторые языки позволяют сделать структуру данных неизменяемой. В этом случае вместо того, чтобы ее инкапсулировать, можно скопировать значение в новое имя, постепенно изменить код пользователей, а затем удалить старое имя. Дублирование данных в случае изменяемых структур данных — это путь к катастрофе; вот почему так популярны неизменяемые данные, устраняющие такие катаклизмы.

Замена вычисленной переменной запросом (Replace Derived Variable with Query)



```
get discountedTotal() {return this._discountedTotal;}
set discount(aNumber) {
  const old = this._discount;
  this._discount = aNumber;
  this._discountedTotal += old - aNumber;
}
```

```
get discountedTotal() {return this._baseTotal - this._discount;}
set discount(aNumber) {this._discount = aNumber;}
```

Мотивация

Одним из крупнейших источников проблем в программном обеспечении являются изменяемые данные. Изменения данных могут часто взаимосвязывать части кода запутанными способами, причем изменения в одной части приводят в другой части к эффектам, которые трудно обнаружить. Во многих ситуациях полностью избежать изменяемых данных нереально, но я рекомендую максимально сократить их количество.

Один из способов, которым я могу оказать значительное влияние на программу, — удалить любые переменные, которые можно легко вычислить. Вычисления часто проясняют смысл данных и защищают от неприятностей, связанных со сбоем обновления переменной при изменении исходных данных.

Разумным исключением из этого правила является случай, когда исходные данные для расчета являются неизменяемыми, и мы можем принудительно сделать неизменяемым результат. Операции преобразования, которые создают новые структуры данных, целесообразно сохранить, даже если они могут быть заменены вычислениями. Фактически здесь имеется дуальность между объектами, которые обертывают структуру данных в ряд вычисляемых свойств, и функциями, которые преобразуют одну структуру данных в другую. Когда исходные данные изменяются и вам приходится управлять временем жизни производных

структур данных, лучше использовать объекты-обертки. Но если исходные данные являются неизменяемыми, или производные данные имеют очень краткое время жизни, то эффективны оба подхода.

Техника

- Определите все точки обновлений переменной. При необходимости используйте рефакторинг *Расщепление переменной* (с. 285) для отделения каждой точки обновления.
- Создайте функцию, которая вычисляет значение переменной.
- Используйте рефакторинг *Введение утверждения* (с. 346), чтобы убедиться, что переменная и вычисление дают один и тот же результат всякий раз при использовании переменной.

  При необходимости создания места для размещения такого утверждения воспользуйтесь рефакторингом *Инкапсуляция переменной* (с. 178).
- Выполните тестирование.
- Замените все чтения переменной вызовом новой функции.
- Выполните тестирование.
- Примените рефакторинг *Удаление неработающего кода* (с. 283) к объявлению и обновлениям переменной.

Пример

Вот отличный маленький пример уродства:

```
class ProductionPlan...
get production() {return this._production;}
applyAdjustment(anAdjustment) {
  this._adjustments.push(anAdjustment);
  this._production += anAdjustment.amount;
}
```

Уродство находится в глазах смотрящего; здесь я вижу его в дублировании — не общем дублировании кода, а дублировании данных. Когда я выполняю `applyAdjustment`, я не только сохраняю настройку, но и использую ее для изменения значения аккумулятора. Я могу просто рассчитать это значение, не обновляя его.

Но я осторожный парень. Моя гипотеза состоит в том, что я могу просто рассчитать ее, и я проверяю эту гипотезу с помощью рефакторинга *Введение утверждения* (с. 346).

```
class ProductionPlan...
get production() {
```

```
  assert(this._production === this.calculatedProduction);
  return this._production;
}
get calculatedProduction() {
  return this._adjustments
    .reduce((sum, a) => sum + a.amount, 0);
}
```

После вставки утверждения я запускаю свои тесты. Если утверждение не сработает, я могу заменить возврат поля возвратом результата вычисления.

```
class ProductionPlan...
get production() {
  assert(this._production === this.calculatedProduction);
  return this.calculatedProduction;
}
```

Теперь я применяю рефакторинг *Встраивание функции* (с. 161):

```
class ProductionPlan...
get production() {
  return this._adjustments
    .reduce((sum, a) => sum + a.amount, 0);
}
```

Я устраняю любые обращения к старой переменной с помощью рефакторинга *Удаление неработающего кода* (с. 283).

```
class ProductionPlan...
applyAdjustment(anAdjustment) {
  this._adjustments.push(anAdjustment);
  this._production += anAdjustment.amount;
}
```

Пример: несколько источников

Приведенный выше пример приятен и прост, потому что ясно, что существует единый источник значения. Но иногда в аккумуляторе может объединяться более одного элемента.

```
class ProductionPlan...
constructor (production) {
  this._production = production;
  this._adjustments = [];
}
get production() {return this._production;}
applyAdjustment(anAdjustment) {
  this._adjustments.push(anAdjustment);
  this._production += anAdjustment.amount;
}
```

Если я выполню тот же рефакторинг *Введение утверждения* (с. 346), что и выше, то теперь он потерпит неудачу в случае ненулевого начального значения `production`.

Но я все еще могу заменить выводимые данные. Разница лишь в том, что сначала я должен применить рефакторинг *Расщепление переменной* (с. 285).

```
constructor (production) {
  this._initialProduction = production;
  this._productionAccumulator = 0;
  this._adjustments = [];
}
get production() {
  return this._initialProduction + this._productionAccumulator;
}
```

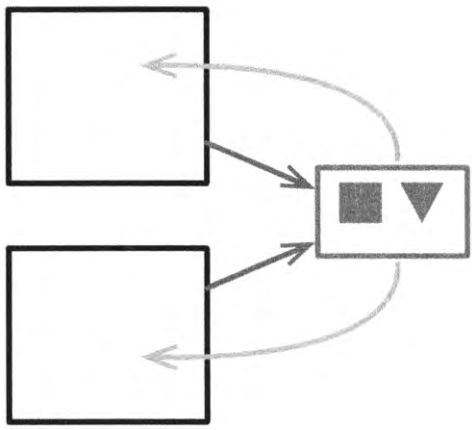
Теперь можно прибегнуть к рефакторингу *Введение утверждения* (с. 346):

```
class ProductionPlan...
get production() {
  assert(this._productionAccumulator ===
         this.calculatedProductionAccumulator);
  return this._initialProduction + this._productionAccumulator;
}
get calculatedProductionAccumulator() {
  return this._adjustments
    .reduce((sum, a) => sum + a.amount, 0);
}
```

и продолжить работу в основном так же, как и раньше. Однако я склонен оставить `totalProductionAjustments` в качестве отдельного свойства, без его встраивания.

Замена ссылки значением (Change Reference to Value)

Обратный к рефакторингу *Замена значения ссылкой* (с. 300)



```
class Product {
  applyDiscount(arg) {this._price.amount -= arg;}
```

```
class Product {
  applyDiscount(arg) {
    this._price =
      new Money(this._price.amount-arg,this._price.currency);
}
```

Мотивация

Вкладывая объект или структуру данных в другую структуру, можно рассматривать внутренний объект как ссылку или как значение. Наиболее очевидна эта разница в том, как обрабатываются обновления свойств внутреннего объекта. Если я рассматриваю его как ссылку, то обновляю свойство внутреннего объекта, сохранив тот же самый внутренний объект. Если же рассматривать его как значение, то можно заменить весь внутренний объект новым, который обладает желаемым свойством.

Рассматривая поле как значение, я могу изменить класс внутреннего объекта, сделав его объектом-значением [39]. С объектами-значениями обычно легче иметь дело потому, что они неизменяемы.

В общем случае с неизменяемыми структурами данных работать легче. Я могу передать неизменяемое значение данных другим частям программы и не беспокоиться о том, что оно может измениться, а объект, в который оно вложено, не узнает об этом изменении. Я могу копировать значения в любом месте программы и не беспокоиться о поддержке ссылок на память. Объекты-значения особенно полезны в распределенных и параллельных системах.

Кроме того, это связано с тем, когда я не должен делать данный рефакторинг. Если я хочу совместно использовать объект с несколькими другими объектами так, чтобы любое изменение в нем было видно всем работающим с ним, то мне нужно, чтобы такой совместно используемый объект был ссылкой.

Техника

- Убедитесь, что класс-кандидат является неизменяемым или может стать таковым.
- Примените рефакторинг *Удаление метода установки значения* (с. 376) к каждому методу установки значения.

- Предоставьте метод проверки равенства на основе значений, который использует поля объекта-значения.

  В большинстве языковых сред для этой цели предусмотрена переопределяемая функция равенства. Обычно вы также должны переопределить метод генератора хеш-кода.

Пример

Представьте, что у нас есть объект человека, который содержит номер телефона.

```
class Person...
constructor() {
  this._telephoneNumber = new TelephoneNumber();
}

get officeAreaCode()    {return this._telephoneNumber.areaCode;}
set officeAreaCode(arg) {this._telephoneNumber.areaCode = arg;}
get officeNumber()    {return this._telephoneNumber.number;}
set officeNumber(arg) {this._telephoneNumber.number = arg;}

class TelephoneNumber...
get areaCode()    {return this._areaCode;}
set areaCode(arg) {this._areaCode = arg;}

get number()    {return this._number;}
set number(arg) {this._number = arg;}
```

Эта ситуация является результатом применения рефакторинга *Извлечение класса* (с. 229), когда старый родительский класс все еще содержит методы обновления нового объекта. Это хорошее время для применения рассматриваемого рефакторинга *Замена ссылки значением*, поскольку существует только одна ссылка на новый класс.

Вначале нужно сделать телефонный номер неизменяемым. Для этого я применяю к полям рефакторинг *Удаление метода установки значения* (с. 376). Первый шаг рефакторинга *Удаление метода установки значения* заключается в использовании рефакторинга *Изменение объявления функции* (с. 170) для добавления двух полей в конструктор и переделки конструктора таким образом, чтобы он вызывал методы установки значений.

```
class TelephoneNumber...
constructor(areaCode, number) {
  this._areaCode = areaCode;
  this._number = number;
}
```

Теперь я смотрю на вызовы методов установки значений. Для каждого из них нужно изменить его на присваивание. Начину с кода города.

```
class Person...
get officeAreaCode()    {return this._telephoneNumber.areaCode;}
set officeAreaCode(arg) {
  this._telephoneNumber=new TelephoneNumber(arg,this.officeNumber);
}
get officeNumber()    {return this._telephoneNumber.number;}
set officeNumber(arg) {this._telephoneNumber.number = arg;}
```

Затем я повторю этот шаг для каждого оставшегося поля.

```
class Person...
get officeAreaCode()    {return this._telephoneNumber.areaCode;}
set officeAreaCode(arg) {
  this._telephoneNumber=new TelephoneNumber(arg,this.officeNumber);
}
get officeNumber()    {return this._telephoneNumber.number;}
set officeNumber(arg) {
  this._telephoneNumber =
    new TelephoneNumber(this.officeAreaCode,arg);
}
```

Теперь телефонный номер неизменяемый и готов стать истинным значением. Проверка того, что объект действительно является значением, заключается в том, что он использует равенство на основе значений. Это область, где JavaScript беспомощен, так как в этом языке и его базовых библиотеках нет ничего, что понимало бы замену равенства, основанного на ссылках, равенством, основанным на значениях. Лучшее, что я могу сделать, — это создать свой собственный метод `equals`.

```
class TelephoneNumber...
equals(other) {
  if (!(other instanceof TelephoneNumber)) return false;
  return this.areaCode === other.areaCode &&
         this.number   === other.number;
}
```

Также важно выполнить проверку наподобие

```
it('telephone equals', function() {
  assert(        new TelephoneNumber("312", "555-0142")
         .equals(new TelephoneNumber("312", "555-0142")));
});
```

Необычное форматирование, которое я здесь использовал, должно подчеркнуть, что это одинаковые вызовы конструктора.

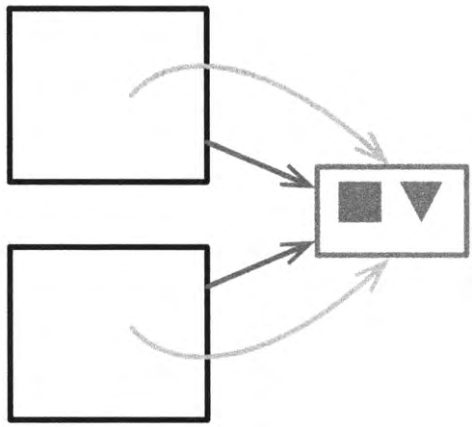
Жизненно важной вещью, которую я делаю в тесте, является создание двух независимых объектов и проверка их равенства.

В большинстве объектно-ориентированных языков программирования есть встроенный тест на равенство, который должен быть переопределен для проверки равенства на основе значений. В Ruby я могу переопределить оператор ==; в Java я переопределяю метод `Object.equals()`. И всякий раз при переопределении метода равенства мне нужно также переопределять метод генерации хеш-кода (например, `Object.hashCode()` в Java), чтобы коллекции, использующие хеширование, правильно работали с новым значением.

Если телефонный номер используется более чем одним клиентом, процедура остается прежней. Применяя рефакторинг *Удаление метода установки значения* (с. 376), я буду модифицировать несколько клиентов вместо одного. Имеют также смысл тесты на неравные телефонные номера, сравнения со значениями, не являющимися телефонными номерами, и с нулевыми значениями.

Замена значения ссылкой (Change Value to Reference)

Обратный к рефакторингу *Замена ссылки значением* (с. 296)



```
let customer = new Customer(customerData);
```

```
let customer = customerRepository.get(customerData.id);
```

Мотивация

Структура данных может иметь несколько записей, связанных с одной и той же логической структурой данных. Я мог бы читать список заказов, часть из которых предназначены для одного и того же клиента. При таком совместном

использовании я могу представить его, рассматривая клиента как значение или как ссылку. В случае значения данные клиента копируются в каждый заказ; при работе со ссылкой существует только одна структура данных, на которую ссылаются несколько заказов.

Если клиент не нуждается в обновлении, то разумны оба подхода. Возможно, немного сбивает с толку наличие нескольких копий одних и тех же данных, но это достаточно распространенная ситуация, не являющаяся особой проблемой. В некоторых случаях из-за наличия нескольких копий могут возникнуть неприятности с памятью, но, как и любая проблема с производительностью, это встречается относительно редко.

Наибольшая трудность при наличии физических копий одних и тех же логических данных возникает, когда мне нужно обновить общие данные. В таком случае я должен найти все копии и все их обновить. Пропустив хотя бы одну, я получу тревожную несогласованность своих данных. В этом случае имеет смысл изменить скопированные данные, используя единственную ссылку. Так любые изменения будут видны всем заказам клиента.

Замена значения ссылкой приводит к тому, что для сущности имеется только один объект, и это обычно означает потребность в каком-то хранилище, где я мог бы получать доступ к этим объектам. Создав единственный объект для сущности, я везде, где он мне нужен, получаю его из этого хранилища.

Техника

- Создайте хранилище для экземпляров связанного объекта (если оно еще не существует).
- Обеспечьте наличие в конструкторе способа поиска правильного экземпляра связанного объекта.
- Измените конструкторы главного объекта так, чтобы для получения связанного объекта использовалось хранилище. Выполните тестирование после каждого изменения.

Пример

Начну с класса, представляющего заказы, которые я могу создавать из входящего документа JSON. Часть данных заказа — идентификатор клиента, на основе которого создается объект клиента.

```
class Order...
constructor(data) {
  this._number = data.number;
  this._customer = new Customer(data.customer);
```

```
  // Загрузка других данных
}
get customer() {return this._customer;}

class Customer...
constructor(id) {
  this._id = id;
}
get id() {return this._id;}
```

Объект клиента, который я создаю таким образом, является значением. Если у меня есть пять заказов, которые ссылаются на идентификатор клиента 123, у меня будет пять отдельных объектов клиента. Любые изменения, которые я внесу в один из них, не будут отражены в других. Если я захочу обогатить объекты клиентов, например, информацией из службы поддержки клиентов, мне придется обновить все пять клиентов одинаковыми данными. Такое наличие дублей объектов всегда заставляет меня нервничать. Несколько объектов, представляющих одну и ту же сущность, например клиент, сбивает с толку. Это особенно неприятно, если объект является изменяемым, что может привести к несогласованности между объектами, представляющими одну и ту же сущность.

Если я хочу использовать каждый раз один и тот же объект клиента, мне нужно место для его хранения. Где именно хранить подобные объекты — решение может (и будет) варьироваться от приложения к приложению, но для простого случая я хотел бы использовать объект репозитория (хранилища) [33].

```
let _repositoryData;

export function initialize() {
  _repositoryData = {};
  _repositoryData.customers = new Map();
}

export function registerCustomer(id) {
  if (! _repositoryData.customers.has(id))
    _repositoryData.customers.set(id, new Customer(id));
  return findCustomer(id);
}

export function findCustomer(id) {
  return _repositoryData.customers.get(id);
}
```

Репозиторий позволяет мне регистрировать объекты клиентов с идентификатором и гарантирует, что с одним и тем же идентификатором создается только один объект клиента. Понятно, что я должен соответствующим образом изменить конструктор заказа, чтобы использовать репозиторий.

Часто при данном рефакторинге хранилище уже существует, поэтому я могу просто его использовать.

Следующий шаг состоит в выяснении, как конструктор заказа может получить правильный объект клиента. В данном случае это просто, поскольку идентификатор клиента присутствует во входном потоке данных.

```
class Order...
constructor(data) {
  this._number = data.number;
  this._customer = registerCustomer(data.customer);
  // Загрузка других данных
}
get customer() {return this._customer;}
```

Теперь любые изменения, которые вносятся в объект клиента одного заказа, будут синхронизированы и видимы всем заказам, принадлежащим этому клиенту.

В данном примере я создаю новый объект клиента при получении первого заказа, который на него ссылается. Другой распространенный подход — получение списка клиентов и заполнение ими хранилища, а затем при чтении заказов можно просто обращаться к уже готовым объектам клиентов. В этом случае заказ, содержащий идентификатор клиента, которого нет в хранилище, будет рассматриваться как ошибка.

Одна из проблем заключается в том, что тело конструктора связано с глобальным хранилищем. К глобальным переменным следует относиться с осторожностью — как сильнодействующее лекарство, они могут быть полезны в малых дозах, и оказаться ядом, если используются слишком часто. В случае неприятностей я могу передавать хранилище конструктору в качестве параметра.

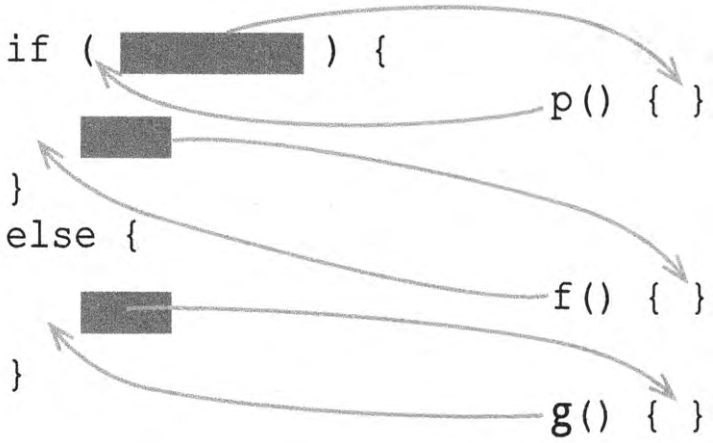
Глава 10

Упрощение условной логики

Большая часть возможностей программ заключается в их способности реализовывать условную логику, но, к сожалению, и большая часть сложности программ заключается именно в этих условных конструкциях. Я часто использую рефакторинг, чтобы облегчить понимание условных конструкций. Я регулярно применяю рефакторинг *Декомпозиция условной инструкции* (с. 306) к сложным условным выражениям и использую рефакторинг *Объединение условного выражения* (с. 309), чтобы сделать комбинации логических операторов более понятными. Я делаю более понятными случаи, когда хочу выполнить некоторые предварительные проверки перед основной обработкой, с помощью рефакторинга *Замена вложенных условных конструкций граничным оператором* (с. 312). Встречая несколько условий, использующих одну и ту же логику переключений, самое время прибегнуть к рефакторингу *Замена условной инструкции полиморфизмом* (с. 317).

Многие условные конструкции используются для обработки частных случаев, таких как нулевые значения; если эта логика в основном одинакова, используется рефакторинг *Введение частного случая* (с. 334) (часто именуемый *Введение нулевого объекта*), который позволяет удалить большое количество дублирующегося кода. И пусть мне очень нравится удалять условные конструкции, но если я хочу сообщить о состоянии программы (и проверить его), то считаю полезным рефакторинг *Введение утверждения* (с. 346).

Декомпозиция условной инструкции (Decompose Conditional)




```
if (!aDate.isBefore(plan.summerStart) &&
    !aDate.isAfter(plan.summerEnd))
  charge = quantity * plan.summerRate;
else
  charge = quantity * plan.regularRate
         + plan.regularServiceCharge;
```

```
if (summer())
  charge = summerCharge();
else
  charge = regularCharge();
```

Мотивация

Одним из наиболее распространенных источников сложности в программе является сложная условная логика. Создавая код для выполнения различных задач в зависимости от разных условий, я могу быстро получить довольно длинную функцию. Длина функции сама по себе является фактором, затрудняющим чтение и понимание, но условия дополнительно усложняют ее. Проблема обычно заключается в том, что код (как в проверке условий, так и в действиях) сообщает о том, что происходит, но может легко скрыть, *почему* это происходит.

Как и в случае любого большого блока кода, я могу прояснить свое намерение, разложив его и заменив каждый фрагмент кода вызовом функции, названным так, чтобы прояснять намерения этого фрагмента. Что касается условий, мне особенно нравится делать это для условной части и каждой альтернативы. Таким образом я подчеркиваю условие и проясняю, что именно делается в ветвях, а также причину этого ветвления.

На самом деле это просто частный случай применения рефакторинга *Извлечение функции* (с. 152) к моему коду, но я предпочитаю выделять этот частный случай, как имеющий особо важное значение.

Техника

- Примените рефакторинг *Извлечение функции* (с. 152) к условию и каждой ветви условной конструкции.

Пример

Предположим, я рассчитываю плату за нечто, что имеет разные тарифы зимой и летом.

```
if (!aDate.isBefore(plan.summerStart) &&
    !aDate.isAfter(plan.summerEnd))
  charge = quantity * plan.summerRate;
else
  charge = quantity * plan.regularRate
         + plan.regularServiceCharge;
```

Выделяю условие в свою собственную функцию.

```
if (summer())
  charge = quantity * plan.summerRate;
else
  charge = quantity * plan.regularRate
         + plan.regularServiceCharge;

function summer() {
  return !aDate.isBefore(plan.summerStart) &&
         !aDate.isAfter(plan.summerEnd);
}
```

После этого обрабатываю ветвь `then`.

```
if (summer())
  charge = summerCharge();
else
  charge = quantity * plan.regularRate
         + plan.regularServiceCharge;
```

```
function summer() {
  return !aDate.isBefore(plan.summerStart) &&
         !aDate.isAfter(plan.summerEnd);
}

function summerCharge() {
  return quantity * plan.summerRate;
}
```

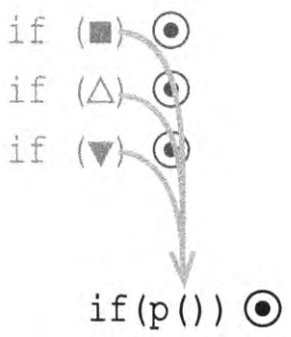
И, наконец, обрабатываю ветвь `else`.

```
if (summer())
  charge = summerCharge();
else
  charge = regularCharge();

function summer() {
  return !aDate.isBefore(plan.summerStart) &&
         !aDate.isAfter(plan.summerEnd);
}

function summerCharge() {
  return quantity * plan.summerRate;
}

function regularCharge() {
  return quantity * plan.regularRate
       + plan.regularServiceCharge;
}
```

По окончании работы я предпочитаю использовать вместо условной конструкции тернарный оператор.

```
charge = summer() ? summerCharge() : regularCharge();

function summer() {
  return !aDate.isBefore(plan.summerStart) &&
         !aDate.isAfter(plan.summerEnd);
}

function summerCharge() {
  return quantity * plan.summerRate;
}

function regularCharge() {
  return quantity * plan.regularRate
       + plan.regularServiceCharge;
}
```

Объединение условного выражения (Consolidate Conditional Expression)




```
if (anEmployee.seniority < 2) return 0;
if (anEmployee.monthsDisabled > 12) return 0;
if (anEmployee.isPartTime) return 0;
```

```
if (isNotEligableForDisability()) return 0;

function isNotEligableForDisability() {
  return ((anEmployee.seniority < 2)
       || (anEmployee.monthsDisabled > 12)
       || (anEmployee.isPartTime));
}
```

Мотивация

Иногда приходится сталкиваться с серией проверок условий, где каждая проверка отлична от других, но результат действия одинаков. Обнаружив такой код, я использую операторы И и ИЛИ, чтобы объединить условия в одно, с одним результатом.

Консолидация условного кода важна по двум причинам. Во-первых, она проясняет ситуацию, показывая, что я действительно делаю только одну проверку, которая объединяет другие проверки. Последовательность дает тот же результат, но выглядит, как будто я выполняю последовательность отдельных проверок, которые просто оказываются близко друг к другу в коде. Вторая причина, по которой я предпочитаю такой рефакторинг, заключается в том, что зачастую он подготавливает меня к рефакторингу *Извлечение функции* (с. 152). Извлечение

условия — одна из самых полезных вещей, которые я могу сделать, чтобы прояснить мой код, так как фактически оказывается заявлением о том, что я делаю и почему.

Аргументы в пользу объединения условий одновременно указывают причины против него. Если я вижу, что это действительно независимые проверки, которые не следует трактовать как проверку единого условия, я отказываюсь от данного рефакторинга.

Техника

- Убедитесь, что ни одно из условий не имеет побочных действий.

  Если такие действия есть — сначала примените к ним рефакторинг *Отделение запроса от модификатора* (с. 352).

- Возьмите две условные инструкции и объедините их условия, используя логический оператор.

  Последовательности объединяются с помощью операторов ИЛИ, вложенные операторы `if` объединяются с помощью операторов И.

- Выполните тестирование.
- Повторяйте объединение условий, пока все они не окажутся собраны в одно-единственное условие.
- Подумайте о применении рефакторинга *Извлечение функции* (с. 152) к получившемуся в результате условию.

Пример

Просматривая некоторый код, я вижу следующее.

```
function disabilityAmount(anEmployee) {
  if (anEmployee.seniority < 2) return 0;
  if (anEmployee.monthsDisabled > 12) return 0;
  if (anEmployee.isPartTime) return 0;
  // Расчет суммы компенсации
```

Это последовательность условных проверок, которые имеют один и тот же результат. Поскольку результат одинаков, я должен объединить эти условия в одно выражение. Для последовательности наподобие данной я использую оператор ИЛИ.

```
function disabilityAmount(anEmployee) {
  if ( (anEmployee.seniority < 2)
    || (anEmployee.monthsDisabled > 12)) return 0;
  if (anEmployee.isPartTime) return 0;
  // Расчет суммы компенсации
```

Я выполняю тестирование, а затем обрабатываю очередное условие.

```
function disabilityAmount(anEmployee) {
  if ( (anEmployee.seniority < 2)
    || (anEmployee.monthsDisabled > 12)
    || (anEmployee.isPartTime)) return 0;
  // Расчет суммы компенсации
```

Собрав все условия вместе, применяю к ним рефакторинг *Извлечение функции* (с. 152).

```
function disabilityAmount(anEmployee) {
  if (isNotEligableForDisability()) return 0;
  // Расчет суммы компенсации

  function isNotEligableForDisability() {
    return ((anEmployee.seniority < 2)
         || (anEmployee.monthsDisabled > 12)
         || (anEmployee.isPartTime));
  }
```

Пример: использование операторов И

В приведенном выше примере показана комбинация условий с помощью операторов ИЛИ, но возможен вариант применения операторов И — при наличии условной конструкции, состоящей из вложенных инструкций `if`.

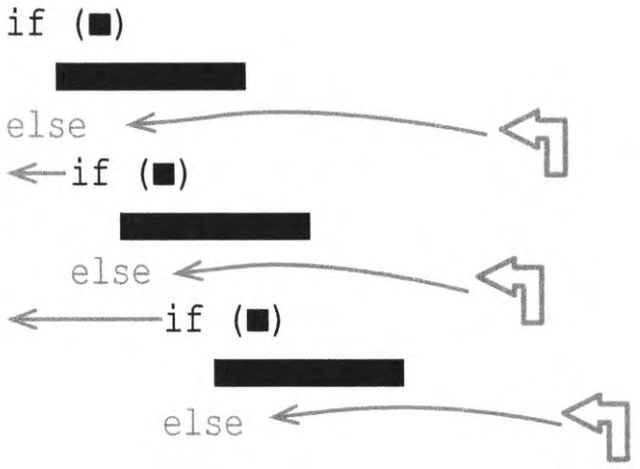
```
if (anEmployee.onVacation)
  if (anEmployee.seniority > 10)
    return 1;
return 0.5;
```

Здесь я выполняю объединение условий с использованием операторов И.

```
if ( (anEmployee.onVacation)
  && (anEmployee.seniority > 10)) return 1;
return 0.5;
```

При сочетании обоих случаев можно по мере необходимости комбинировать операторы И и ИЛИ. В этой ситуации достаточно легко запутаться, поэтому я использую рефакторинг *Извлечение функции* (с. 152), чтобы сделать код понятным.

Замена вложенных условных конструкций граничным оператором (Replace Nested Conditional with Guard Clauses)




```
function getPayAmount() {
  let result;
  if (isDead)
    result = deadAmount();
  else {
    if (isSeparated)
      result = separatedAmount();
    else {
      if (isRetired)
        result = retiredAmount();
      else
        result = normalPayAmount();
    }
  }
  return result;
}
```

```
function getPayAmount() {
  if (isDead) return deadAmount();
  if (isSeparated) return separatedAmount();
  if (isRetired) return retiredAmount();
  return normalPayAmount();
}
```

Мотивация

Условные выражения чаще всего имеют один из двух видов. В первом случае это проверка, при которой любая выбранная альтернатива является частью нормального поведения. Во втором случае мы имеем дело с ситуацией, когда одна альтернатива условной инструкции представляет собой нормальное поведение, а вторая — некоторые необычные условия.

Эти виды условных инструкций имеют разное назначение, и оно должно быть очевидно из кода. Если обе части представляют собой нормальное поведение, используйте условие с ветвями `if` и `else`. Если же некоторое условие является необычным, то если его проверка указывает истинность условия, выполняется `return`. Такого рода проверка часто называется **граничным оператором** (guard clause).

Главный смысл описываемого здесь рефакторинга — в придании коду выразительности. При использовании конструкции `if-then-else` ветви `then` и `else` получают равный вес. Это говорит читателю кода, что обе ветви обладают равными вероятностью и важностью. Граничный же оператор гласит: "Такое происходит редко, и если оно все же произошло, то надо выполнить вот такие действия и выйти".

Я часто прибегаю к этому рефакторингу, когда работаю с программистом, которого учили, что в методе должны быть единственная точка входа и единственная точка выхода. Единственная точка входа обеспечивается самими современными языками программирования, но в правиле единственной точки выхода на самом деле никакой пользы нет. Главным принципом должна быть ясность: если метод понятнее, когда в нем одна точка выхода, используйте ее; в противном же случае не следует стремиться к этому любой ценой.

Техника

- Выберите условие верхнего уровня, которое необходимо заменить, и превратите его в граничный оператор.
- Выполните тестирование.
- При необходимости повторите эти действия.
- Если все граничные операторы возвращают один и тот же результат, воспользуйтесь рефакторингом *Объединение условного выражения* (с. 309).

Пример

Вот некоторый код для расчета суммы оплаты сотрудника. Он актуален только в том случае, когда сотрудник работает в компании, поэтому код должен проверить два других случая.

```
function payAmount(employee) {
  let result;
  if(employee.isSeparated) {
    result = {amount: 0, reasonCode: "SEP"};
  }
  else {
    if (employee.isRetired) {
      result = {amount: 0, reasonCode: "RET"};
    }
    else {
      // Логика вычисления оплаты
      lorem.ipsum(dolor.sitAmet);
      consectetur(adipiscing).elit();
      sed.do.eiusmod =
        tempor.incididunt.ut(labore) && dolore(magna.aliqua);
      ut.enim.ad(minim.veniam);
      result = someFinalComputation();
    }
  }
  return result;
}
```

Здесь вложенность условных выражений маскирует истинный смысл происходящего. Основная цель этого кода работает только в том случае, если все эти условия ложны. В такой ситуации предназначение кода оказывается более ясным при применении граничных операторов. Как и в случае любого рефакторинга, я предпочитаю идти небольшими шагами, поэтому начинаю с верхнего условия.

```
function payAmount(employee) {
  let result;
  if (employee.isSeparated) return {amount: 0, reasonCode: "SEP"};
  if (employee.isRetired) {
    result = {amount: 0, reasonCode: "RET"};
  }
  else {
    // Логика вычисления оплаты
    lorem.ipsum(dolor.sitAmet);
    consectetur(adipiscing).elit();
    sed.do.eiusmod =
      tempor.incididunt.ut(labore) && dolore(magna.aliqua);
    ut.enim.ad(minim.veniam);
    result = someFinalComputation();
  }
  return result;
}
```

Я выполняю тестирование данного изменения и перехожу к следующему.

```
function payAmount(employee) {
  let result;
  if (employee.isSeparated) return {amount: 0, reasonCode: "SEP"};
  if (employee.isRetired) return {amount: 0, reasonCode: "RET"};
  // Логика вычисления оплаты
  lorem.ipsum(dolor.sitAmet);
  consectetur(adipiscing).elit();
  sed.do.eiusmod =
    tempor.incididunt.ut(labore) && dolore(magna.aliqua);
  ut.enim.ad(minim.veniam);
  result = someFinalComputation();
  return result;
}
```

В этот момент переменная результата на самом деле не делает ничего полезного, поэтому я удаляю ее.

```
function payAmount(employee) {
  let result;
  if (employee.isSeparated) return {amount: 0, reasonCode: "SEP"};
  if (employee.isRetired) return {amount: 0, reasonCode: "RET"};
  // Логика вычисления оплаты
  lorem.ipsum(dolor.sitAmet);
  consectetur(adipiscing).elit();
  sed.do.eiusmod =
    tempor.incididunt.ut(labore) && dolore(magna.aliqua);
  ut.enim.ad(minim.veniam);
  return someFinalComputation();
}
```

Удаление изменяемой переменной — это всегда хорошо.

Пример: обращение условий

Рецензируя рукопись этой книги, Джошуа Кериевски (Joshua Kerievsky) заметил, что зачастую данный рефакторинг выполняется с обращением условных выражений. Он любезно предоставил пример, который позволил мне не перенапрягать собственное воображение.

```
function adjustedCapital(anInstrument) {
  let result = 0;
  if (anInstrument.capital > 0) {
    if (anInstrument.interestRate > 0 &&
        anInstrument.duration     > 0) {
      result = (anInstrument.income / anInstrument.duration)
            * anInstrument.adjustmentFactor;
    }
  }
  return result;
}
```

Я снова поочередно выполняю замену, но на этот раз при вставке граничного оператора условие делается обратным:

```
function adjustedCapital(anInstrument) {
  let result = 0;
  if (anInstrument.capital <= 0) return result;
  if (anInstrument.interestRate > 0 &&
      anInstrument.duration     > 0) {
    result = (anInstrument.income / anInstrument.duration)
           * anInstrument.adjustmentFactor;
  }
  return result;
}
```

Следующее условие немного сложнее, поэтому я работаю с ним в два этапа. Сначала я просто добавляю `НЕ`.

```
function adjustedCapital(anInstrument) {
  let result = 0;
  if (anInstrument.capital <= 0) return result;
  if (!(anInstrument.interestRate>0 && anInstrument.duration>0))
    return result;
  result = (anInstrument.income / anInstrument.duration)
         * anInstrument.adjustmentFactor;
  return result;
}
```

Оставляя `НЕ` в условном выражении наподобие данного, можно исказить его восприятие, поэтому я упрощаю его.

```
function adjustedCapital(anInstrument) {
  let result = 0;
  if (anInstrument.capital <= 0) return result;
  if (anInstrument.interestRate <= 0 || anInstrument.duration <= 0)
    return result;
  result = (anInstrument.income / anInstrument.duration)
         * anInstrument.adjustmentFactor;
  return result;
}
```

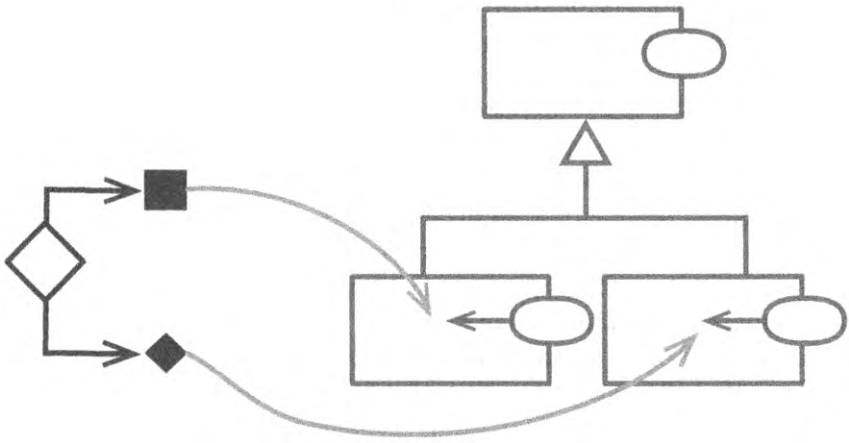
Обе эти строки содержат условия с одинаковым результатом, поэтому я применяю рефакторинг *Объединение условного выражения* (с. 309).

```
function adjustedCapital(anInstrument) {
  let result = 0;
  if ( anInstrument.capital <= 0
    || anInstrument.interestRate <= 0
    || anInstrument.duration <= 0) return result;
  result = (anInstrument.income / anInstrument.duration)
         *  anInstrument.adjustmentFactor;
  return result;
}
```

Переменная `result` играет здесь двойную роль. Первое присваивание нулевого значения этой переменной указывает, что следует возвратить, когда срабатывает граничный оператор; второе его значение — результат окончательного вычисления. Я могу избавиться от него, что исключит двойное использование переменной.

```
function adjustedCapital(anInstrument) {
  if ( anInstrument.capital <= 0
    || anInstrument.interestRate <= 0
    || anInstrument.duration <= 0) return 0;
  return (anInstrument.income / anInstrument.duration)
        * anInstrument.adjustmentFactor;
}
```

Замена условной инструкции полиморфизмом (Replace Conditional with Polymorphism)



```
switch (bird.type) {
  case 'EuropeanSwallow':
    return "average";
  case 'AfricanSwallow':
    return (bird.numberOfCoconuts > 2) ? "tired" : "average";
  case 'NorwegianBlueParrot':
    return (bird.voltage > 100) ? "scorched" : "beautiful";
  default:
    return "unknown";
```

```
class EuropeanSwallow {
  get plumage() {
    return "average";
  }
class AfricanSwallow {
  get plumage() {
    return (this.numberOfCoconuts > 2) ? "tired" : "average";
  }
class NorwegianBlueParrot {
  get plumage() {
    return (this.voltage > 100) ? "scorched" : "beautiful";
  }
```

Мотивация

Сложная условная логика — одна из самых трудных вещей в программировании, поэтому я всегда ищу способы добавить к ней структуру. Часто выясняется, что для разделения условий можно разделить логику на различные случаи высокого уровня. Иногда достаточно представить это разделение в структуре самого условия, но использование классов и полиморфизма может сделать разделение более явным.

Распространенным является случай, когда можно сформировать набор типов, каждый из которых обрабатывает условную логику по-своему. Очевидно, что обработка книг, музыки и еды различается в зависимости от их типа. Наиболее заметным это становится тогда, когда есть несколько функций с инструкцией `switch` для кода типа. В этом случае я удаляю дублирование общей логики `switch`, создавая для каждого случая классы и используя полиморфизм для создания поведения, специфичного для типа.

Еще одна ситуация — когда логику можно представить как базовый случай с вариациями. Этот базовый случай может быть самым распространенным или самым простым. Я могу поместить данную базовую логику в суперкласс, что позволит мне работать с ним, не беспокоясь о вариациях. Затем я помещаю каждый вариант `case` в подкласс, который выражен с помощью кода, подчеркивающего его отличие от базового варианта.

Полиморфизм является одной из ключевых особенностей объектно-ориентированного программирования, и, как и любая полезная функциональная возможность, он подвержен риску злоупотребления. Я встречал людей, которые утверждают, что следует заменить полиморфизмом все случаи условной логики. Я не согласен с этим мнением. Большая часть моей условной логики использует обычные конструкции `if/else` и `switch/case`. Но когда я вижу сложную условную логику, которую можно улучшить рассмотренным выше способом, я считаю полиморфизм подходящим мощным инструментом.

Техника

- Если классы с полиморфным поведением не существуют, создайте их вместе с фабричной функцией, возвращающей правильный экземпляр.
- Используйте фабричную функцию в вызывающем коде.
- Переместите условную функцию в суперкласс.

  Если условная логика не представлена автономной функцией — воспользуйтесь рефакторингом *Извлечение функции* (с. 152).
- Выберите один из подклассов. Создайте метод подкласса, который перекрывает метод условной инструкции. Скопируйте тело этой ветви условной инструкции в метод подкласса и настройте его для корректной работы в составе подкласса.
- Повторите эти действия для каждой ветви условной конструкции.
- Оставьте случай для метода суперкласса. Или, если суперкласс должен быть абстрактным, объявите этот метод как абстрактный или сгенерируйте ошибку, чтобы показать, что он должен быть ответственностью подкласса.

Пример

У моего друга есть коллекция птиц, и он хочет знать, как быстро они могут летать и какое у них оперение. Итак, у нас есть пара небольших программ для получения нужной информации.

```
function plumages(birds) {
  return new Map(birds.map(b => [b.name, plumage(b)]));
}

function speeds(birds) {
  return new Map(birds.map(b => [b.name, airSpeedVelocity(b)]));
}

function plumage(bird) {
  switch (bird.type) {
  case 'EuropeanSwallow':
    return "average";
  case 'AfricanSwallow':
    return (bird.numberOfCoconuts > 2) ? "tired" : "average";
  case 'NorwegianBlueParrot':
    return (bird.voltage > 100) ? "scorched" : "beautiful";
  default:
    return "unknown";
  }
}
```

```
function airSpeedVelocity(bird) {
  switch (bird.type) {
  case 'EuropeanSwallow':
    return 35;
  case 'AfricanSwallow':
    return 40 - 2 * bird.numberOfCoconuts;
  case 'NorwegianBlueParrot':
    return (bird.isNailed) ? 0 : 10 + bird.voltage / 10;
  default:
    return null;
  }
}
```

Существуют две различные операции, которые различаются в зависимости от вида птиц, поэтому имеет смысл создать классы и использовать полиморфизм для специфичного типа поведения.

Я начинаю с применения рефакторинга *Объединение функций в класс* (с. 190) к `airSpeedVelocity` и `plumage`.

```
function plumage(bird) {
  return new Bird(bird).plumage;
}

function airSpeedVelocity(bird) {
  return new Bird(bird).airSpeedVelocity;
}

class Bird {
  constructor(birdObject) {
    Object.assign(this, birdObject);
  }
  get plumage() {
    switch (this.type) {
    case 'EuropeanSwallow':
      return "average";
    case 'AfricanSwallow':
      return (this.numberOfCoconuts > 2) ? "tired" : "average";
    case 'NorwegianBlueParrot':
      return (this.voltage > 100) ? "scorched" : "beautiful";
    default:
      return "unknown";
    }
  }

  get airSpeedVelocity() {
    switch (this.type) {
    case 'EuropeanSwallow':
      return 35;
    case 'AfricanSwallow':
      return 40 - 2 * this.numberOfCoconuts;
```

```
    case 'NorwegianBlueParrot':
      return (this.isNailed) ? 0 : 10 + this.voltage / 10;
    default:
      return null;
    }
  }
}
```

Теперь добавляю подклассы для каждого вида птиц вместе с фабричной функцией, чтобы создать экземпляр соответствующего подкласса.

```
function plumage(bird) {
  return createBird(bird).plumage;
}

function airSpeedVelocity(bird) {
  return createBird(bird).airSpeedVelocity;
}

function createBird(bird) {
  switch (bird.type) {
  case 'EuropeanSwallow':
    return new EuropeanSwallow(bird);
  case 'AfricanSwallow':
    return new AfricanSwallow(bird);
  case 'NorweigianBlueParrot':
    return new NorwegianBlueParrot(bird);
  default:
    return new Bird(bird);
  }
}

class EuropeanSwallow extends Bird {
}

class AfricanSwallow extends Bird {
}

class NorwegianBlueParrot extends Bird {
}
```

Создав нужную структуру классов, можно воспользоваться двумя условными методами. Начну с оперения. Я беру одну ветвь инструкции `switch` и перекрываю ее в соответствующем подклассе.

```
class EuropeanSwallow...
get plumage() {
  return "average";
}
class Bird...
get plumage() {
```

```
  switch (this.type) {
  case 'EuropeanSwallow':
    throw "oops";
  case 'AfricanSwallow':
    return (this.numberOfCoconuts > 2) ? "tired" : "average";
  case 'NorwegianBlueParrot':
    return (this.voltage > 100) ? "scorched" : "beautiful";
  default:
    return "unknown";
  }
}
```

Я помещаю `throw` в силу своего параноидального характера.

В этот момент можно выполнить компиляцию и тестирование. Затем, если все в порядке, я перехожу к следующей ветви.

```
class AfricanSwallow...
get plumage() {
  return (this.numberOfCoconuts > 2) ? "tired" : "average";
}
```

И к еще одной:

```
class NorwegianBlueParrot...
get plumage() {
  return (this.voltage > 100) ? "scorched" : "beautiful";
}
```

Метод суперкласса остается для ветви `default`.

```
class Bird...
get plumage() {
  return "unknown";
}
```

Затем я повторяю этот процесс для `airSpeedVelocity`. По окончании работы получается следующий код (я также выполнил встраивание функций верхнего уровня для `airSpeedVelocity` и `plumage`).

```
function plumages(birds) {
  return new Map(birds
                 .map(b => createBird(b))
                 .map(bird => [bird.name, bird.plumage]));
}

function speeds(birds) {
  return new Map(birds
                 .map(b => createBird(b))
                 .map(bird => [bird.name, bird.airSpeedVelocity]));
}
```

```
function createBird(bird) {
  switch (bird.type) {
  case 'EuropeanSwallow':
    return new EuropeanSwallow(bird);
  case 'AfricanSwallow':
    return new AfricanSwallow(bird);
  case 'NorwegianBlueParrot':
    return new NorwegianBlueParrot(bird);
  default:
    return new Bird(bird);
  }
}

class Bird {
  constructor(birdObject) {
    Object.assign(this, birdObject);
  }
  get plumage() {
    return "unknown";
  }
  get airSpeedVelocity() {
    return null;
  }
}

class EuropeanSwallow extends Bird {
  get plumage() {
    return "average";
  }
  get airSpeedVelocity() {
    return 35;
  }
}

class AfricanSwallow extends Bird {
  get plumage() {
    return (this.numberOfCoconuts > 2) ? "tired" : "average";
  }
  get airSpeedVelocity() {
    return 40 - 2 * this.numberOfCoconuts;
  }
}

class NorwegianBlueParrot extends Bird {
  get plumage() {
    return (this.voltage > 100) ? "scorched" : "beautiful";
  }
  get airSpeedVelocity() {
  return (this.isNailed) ? 0 : 10 + this.voltage / 10;
  }
}
```

Глядя на этот окончательный код, я вижу, что суперкласс `Bird` не так уж нужен. В JavaScript для полиморфизма мне не нужна иерархия типов; до тех пор, пока мои объекты реализуют методы с соответствующими именами, все работает нормально. Однако в подобной ситуации я предпочитаю оставлять ненужный суперкласс, поскольку он помогает объяснить, как связаны используемые классы.

Пример: использование полиморфизма для вариативного поведения

В примере с птицами я использовал ясную иерархию обобщения. Именно так в учебниках (включая написанные мной) часто рассматриваются подклассы и полиморфизм, но это не единственный способ использования наследования на практике. Более того, это, вероятно, не самый распространенный или хороший способ. Другой случай наследования — когда я хочу указать, что один объект в основном похож на другой, но с определенными вариациями.

В качестве примера этого случая рассмотрим некоторый код, используемый рейтинговым агентством для расчета инвестиционного рейтинга для рейсов парусных судов. Рейтинговое агентство присваивает рейтинг *A* или *B* в зависимости от различных факторов, связанных с риском и потенциальной прибылью. Риск исходит из оценки характера рейса, а также истории предыдущих рейсов капитана.

```
function rating(voyage, history) {
  const vpf = voyageProfitFactor(voyage, history);
  const vr = voyageRisk(voyage);
  const chr = captainHistoryRisk(voyage, history);
  if (vpf * 3 > (vr + chr * 2)) return "A";
  else return "B";
}

function voyageRisk(voyage) {
  let result = 1;
  if (voyage.length > 4) result += 2;
  if (voyage.length > 8) result += voyage.length - 8;
  if (["china", "east-indies"].includes(voyage.zone)) result += 4;
  return Math.max(result, 0);
}

function captainHistoryRisk(voyage, history) {
  let result = 1;
  if (history.length < 5) result += 4;
  result += history.filter(v => v.profit < 0).length;
  if (voyage.zone === "china" && hasChina(history)) result -= 2;
  return Math.max(result, 0);
}

function hasChina(history) {
  return history.some(v => "china" === v.zone);
}
```

```
function voyageProfitFactor(voyage, history) {
  let result = 2;
  if (voyage.zone === "china") result += 1;
  if (voyage.zone === "east-indies") result += 1;
  if (voyage.zone === "china" && hasChina(history)) {
    result += 3;
    if (history.length > 10) result += 1;
    if (voyage.length > 12) result += 1;
    if (voyage.length > 18) result -= 1;
  }
  else {
    if (history.length > 8) result += 1;
    if (voyage.length > 14) result -= 1;
  }
  return result;
}
```

Функции `voyageRisk` и `captainHistoryRisk` оценивают баллы за риск, `voyageProfitFactor` учитывает баллы за потенциальную прибыль, а `rating` объединяет их, чтобы дать общую оценку рейтинга рейса.

Вызывающий код выглядит примерно следующим образом.

```
const voyage = {zone: "west-indies", length: 10};
const history = [
  {zone: "east-indies", profit: 5},
  {zone: "west-indies", profit: 15},
  {zone: "china", profit: -2},
  {zone: "west-africa", profit: 7},
];
const myRating = rating(voyage, history);
```

Здесь я хочу сосредоточиться на том, как пара мест использует условную логику для рейса в Китай, если капитан бывал в Китае раньше.

```
function rating(voyage, history) {
  const vpf = voyageProfitFactor(voyage, history);
  const vr = voyageRisk(voyage);
  const chr = captainHistoryRisk(voyage, history);
  if (vpf * 3 > (vr + chr * 2)) return "A";
  else return "B";
}

function voyageRisk(voyage) {
  let result = 1;
  if (voyage.length > 4) result += 2;
  if (voyage.length > 8) result += voyage.length - 8;
  if (["china", "east-indies"].includes(voyage.zone)) result += 4;
  return Math.max(result, 0);
}

function captainHistoryRisk(voyage, history) {
  let result = 1;
```

```
  if (history.length < 5) result += 4;
  result += history.filter(v => v.profit < 0).length;
  if (voyage.zone === "china" && hasChina(history)) result -= 2;
  return Math.max(result, 0);
}

function hasChina(history) {
  return history.some(v => "china" === v.zone);
}

function voyageProfitFactor(voyage, history) {
  let result = 2;
  if (voyage.zone === "china") result += 1;
  if (voyage.zone === "east-indies") result += 1;
  if (voyage.zone === "china" && hasChina(history)) {
    result += 3;
    if (history.length > 10) result += 1;
    if (voyage.length > 12) result += 1;
    if (voyage.length > 18) result -= 1;
  }
  else {
    if (history.length > 8) result += 1;
    if (voyage.length > 14) result -= 1;
  }
  return result;
}
```

Я буду использовать наследование и полиморфизм, чтобы отделить логику обработки этих случаев от базовой логики. Этот рефакторинг особенно полезен, если я собираюсь ввести некоторую особую логику для данного случая, и эта логика для повторных рейсов в Китай может усложнить понимание базового случая.

Я начинаю с множества функций. Чтобы использовать полиморфизм, нужно создать структуру классов, так что начну с применения рефакторинга *Объединение функций в класс* (с. 190). Он приводит к следующему коду.

```
function rating(voyage, history) {
  return new Rating(voyage, history).value;
}

class Rating {
  constructor(voyage, history) {
    this.voyage = voyage;
    this.history = history;
  }
  get value() {
    const vpf = this.voyageProfitFactor;
    const vr = this.voyageRisk;
    const chr = this.captainHistoryRisk;
    if (vpf * 3 > (vr + chr * 2)) return "A";
    else return "B";
  }
```

```
  get voyageRisk() {
    let result = 1;
    if (this.voyage.length > 4) result += 2;
    if (this.voyage.length > 8) result += this.voyage.length - 8;
    if (["china", "east-indies"].includes(this.voyage.zone))
      result += 4;
    return Math.max(result, 0);
  }
  get captainHistoryRisk() {
    let result = 1;
    if (this.history.length < 5) result += 4;
    result += this.history.filter(v => v.profit < 0).length;
    if (this.voyage.zone === "china" && this.hasChinaHistory)
      result -= 2;
    return Math.max(result, 0);
  }
  get voyageProfitFactor() {
    let result = 2;

    if (this.voyage.zone === "china") result += 1;
    if (this.voyage.zone === "east-indies") result += 1;
    if (this.voyage.zone === "china" && this.hasChinaHistory) {
      result += 3;
      if (this.history.length > 10) result += 1;
      if (this.voyage.length > 12) result += 1;
      if (this.voyage.length > 18) result -= 1;
    }
    else {
      if (this.history.length > 8) result += 1;
      if (this.voyage.length > 14) result -= 1;
    }
    return result;
  }
  get hasChinaHistory() {
    return this.history.some(v => "china" === v.zone);
  }
}
```

Этот код дает класс для базового случая. Теперь следует создать пустой подкласс для размещения варианта поведения.

```
class ExperiencedChinaRating extends Rating {
}
```

Затем я создаю фабричную функцию, которая при необходимости возвращает нужный вариант класса.

```
function createRating(voyage, history) {
  if (voyage.zone === "china" &&
      history.some(v => "china" === v.zone))
    return new ExperiencedChinaRating(voyage, history);
  else return new Rating(voyage, history);
}
```

Следует изменить все вызовы таким образом, чтобы они использовали фабричную функцию вместо прямого вызова конструктора.

```
function rating(voyage, history) {
  return createRating(voyage, history).value;
}
```

Есть два фрагмента поведения, которые нужно переместить в подкласс. Начну с логики в `captainHistoryRisk`.

```
class Rating...
get captainHistoryRisk() {
  let result = 1;
  if (this.history.length < 5) result += 4;
  result += this.history.filter(v => v.profit < 0).length;
  if (this.voyage.zone === "china" && this.hasChinaHistory)
    result -= 2;
  return Math.max(result, 0);
}
```

Напишу перекрывающий метод в подклассе.

```
class ExperiencedChinaRating...
get captainHistoryRisk() {
  const result = super.captainHistoryRisk - 2;
  return Math.max(result, 0);
}

class Rating...
get captainHistoryRisk() {
  let result = 1;
  if (this.history.length < 5) result += 4;
  result += this.history.filter(v => v.profit < 0).length;
  if (this.voyage.zone === "china" && this.hasChinaHistory)
    result -= 2;
  return Math.max(result, 0);
}
```

Выделение вариативного поведения из `voyageProfitFactor` немного тяжелее. Нельзя просто удалить вариант поведения и вызвать метод суперкласса, так как здесь имеется альтернативный путь. Я также не хочу копировать весь метод суперкласса в подкласс.

```
class Rating...
get voyageProfitFactor() {
  let result = 2;

  if (this.voyage.zone === "china") result += 1;
  if (this.voyage.zone === "east-indies") result += 1;
  if (this.voyage.zone === "china" && this.hasChinaHistory) {
    result += 3;
```

```
    if (this.history.length > 10) result += 1;
    if (this.voyage.length > 12) result += 1;
    if (this.voyage.length > 18) result -= 1;
  }
  else {
    if (this.history.length > 8) result += 1;
    if (this.voyage.length > 14) result -= 1;
  }
  return result;
}
```

Поэтому решение состоит в том, чтобы сначала использовать рефакторинг *Извлечение функции* (с. 152) для всего условного блока.

```
class Rating...
get voyageProfitFactor() {
  let result = 2;

  if (this.voyage.zone === "china") result += 1;
  if (this.voyage.zone === "east-indies") result += 1;
  result += this.voyageAndHistoryLengthFactor;
  return result;
}

get voyageAndHistoryLengthFactor() {
  let result = 0;
  if (this.voyage.zone === "china" && this.hasChinaHistory) {
    result += 3;
    if (this.history.length > 10) result += 1;
    if (this.voyage.length > 12) result += 1;
    if (this.voyage.length > 18) result -= 1;
  }
  else {
    if (this.history.length > 8) result += 1;
    if (this.voyage.length > 14) result -= 1;
  }
  return result;
}
```

Имя функции с *И* (And) посредине — неприятный запах, но я позволю ему немного попахнуть, пока я создаю подклассы.

```
class Rating...
get voyageAndHistoryLengthFactor() {
  let result = 0;
  if (this.history.length > 8) result += 1;
  if (this.voyage.length > 14) result -= 1;
  return result;
}
class ExperiencedChinaRating...
get voyageAndHistoryLengthFactor() {
```

```
  let result = 0;
  result += 3;
  if (this.history.length > 10) result += 1;
  if (this.voyage.length > 12) result += 1;
  if (this.voyage.length > 18) result -= 1;
  return result;
}
```

Формально рефакторинг на этом завершен — я разделил вариативное поведение на подклассы. Логика суперкласса проще для понимания и работы, и когда я работаю над кодом подкласса, мне нужно иметь дело только с отличиями данного варианта от основной логики, которые выражаются как отличие подкласса от суперкласса.

Но я чувствую, что должен обрисовать хотя бы в общих чертах, что буду делать с неуклюжим новым методом. Введение метода, предназначенного исключительно для перекрытия подклассами, является обычным при выполнении такого рода наследования, основанного на отличиях поведения подклассов и базового класса. Увы, грубый метод, подобный описанному, только затемняет происходящее, вместо того чтобы его раскрывать.

Наличие *И* (And) в имени функции говорит о том, что в действительности здесь выполняются две отдельные модификации. Я думаю, что было бы разумно разделить их с помощью рефакторинга *Извлечение функции* (с. 152). Начну с суперкласса.

```
class Rating...
get voyageAndHistoryLengthFactor() {
  let result = 0;
  result += this.historyLengthFactor;
  if (this.voyage.length > 14) result -= 1;
  return result;
}
get historyLengthFactor() {
  return (this.history.length > 8) ? 1 : 0;
}
```

Затем выполню то же с подклассом.

```
class ExperiencedChinaRating...
get voyageAndHistoryLengthFactor() {
let result = 0;
  result += 3;
  result += this.historyLengthFactor;
  if (this.voyage.length > 12) result += 1;
  if (this.voyage.length > 18) result -= 1;
  return result;
}

get historyLengthFactor() {
  return (this.history.length > 10) ? 1 : 0;
}
```

Теперь к суперклассу можно применить рефакторинг *Перенос инструкций в точку вызова* (с. 262).

```
class Rating...
get voyageProfitFactor() {
  let result = 2;
  if (this.voyage.zone === "china") result += 1;
  if (this.voyage.zone === "east-indies") result += 1;
  result += this.historyLengthFactor;
  result += this.voyageAndHistoryLengthFactor;
  return result;
}

get voyageAndHistoryLengthFactor() {
  let result = 0;
  result += this.historyLengthFactor;
  if (this.voyage.length > 14) result -= 1;
  return result;
}

class ExperiencedChinaRating...
get voyageAndHistoryLengthFactor() {
  let result = 0;
  result += 3;
  result += this.historyLengthFactor;
  if (this.voyage.length > 12) result += 1;
  if (this.voyage.length > 18) result -= 1;
  return result;
}
```

Затем я бы использовал рефакторинг *Переименование функции* (с. 170).

```
class Rating...
get voyageProfitFactor() {
  let result = 2;
  if (this.voyage.zone === "china") result += 1;
  if (this.voyage.zone === "east-indies") result += 1;
  result += this.historyLengthFactor;
  result += this.voyageLengthFactor;
  return result;
}

get voyageLengthFactor() {
  return (this.voyage.length > 14) ? - 1: 0;
}
```

Здесь я применил тернарный оператор для упрощения `voyageLengthFactor`.

```
class ExperiencedChinaRating...
get voyageLengthFactor() {
  let result = 0;
  result += 3;
```

```
  if (this.voyage.length > 12) result += 1;
  if (this.voyage.length > 18) result -= 1;
  return result;
}
```

И последнее. Я не думаю, что добавление 3 баллов имеет смысл в качестве части коэффициента продолжительности рейса — их лучше добавлять к общему результату.

```
class ExperiencedChinaRating...
get voyageProfitFactor() {
  return super.voyageProfitFactor + 3;
}

get voyageLengthFactor() {
  let result = 0;
  result += 3;
  if (this.voyage.length > 12) result += 1;
  if (this.voyage.length > 18) result -= 1;
  return result;
}
```

В конце рефакторинга я получаю следующий код. Во-первых, имеется базовый класс рейтинга, который может игнорировать любые сложности, связанные с Китаем.

```
class Rating {
  constructor(voyage, history) {
    this.voyage = voyage;
    this.history = history;
  }
  get value() {
    const vpf = this.voyageProfitFactor;
    const vr = this.voyageRisk;
    const chr = this.captainHistoryRisk;
    if (vpf * 3 > (vr + chr * 2)) return "A";
    else return "B";
  }
  get voyageRisk() {
    let result = 1;
    if (this.voyage.length > 4) result += 2;
    if (this.voyage.length > 8) result += this.voyage.length - 8;
    if (["china", "east-indies"].includes(this.voyage.zone)) result +=
4;
    return Math.max(result, 0);
  }
```

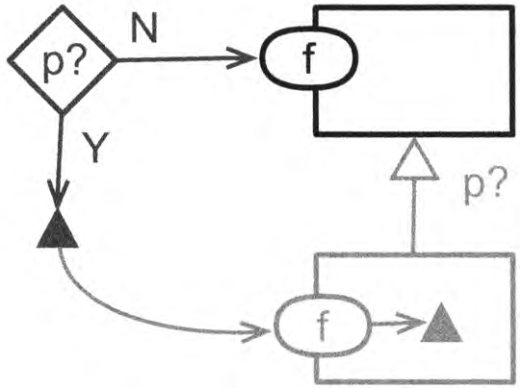
```
  get captainHistoryRisk() {
    let result = 1;
    if (this.history.length < 5) result += 4;
    result += this.history.filter(v => v.profit < 0).length;
    return Math.max(result, 0);
  }
  get voyageProfitFactor() {
    let result = 2;
    if (this.voyage.zone === "china") result += 1;
    if (this.voyage.zone === "east-indies") result += 1;
    result += this.historyLengthFactor;
    result += this.voyageLengthFactor;
    return result;
  }
  get voyageLengthFactor() {
    return (this.voyage.length > 14) ? - 1: 0;
  }
  get historyLengthFactor() {
    return (this.history.length > 8) ? 1 : 0;
  }
}
```

Код, связанный с наличием опыта плаваний в Китай, читается как множество вариаций базового класса.

```
class ExperiencedChinaRating extends Rating {
  get captainHistoryRisk() {
    const result = super.captainHistoryRisk - 2;
    return Math.max(result, 0);
  }
  get voyageLengthFactor() {
    let result = 0;
    if (this.voyage.length > 12) result += 1;
    if (this.voyage.length > 18) result -= 1;
    return result;
  }
  get historyLengthFactor() {
    return (this.history.length > 10) ? 1 : 0;
  }
  get voyageProfitFactor() {
    return super.voyageProfitFactor + 3;
  }
}
```

Введение частного случая (Introduce Special Case)

Бывший рефакторинг *Введение нулевого объекта*




```
if (aCustomer === "unknown") customerName = "occupant";
```

```
class UnknownCustomer {
  get name() {return "occupant";}
```

Мотивация

Распространенным случаем дублированного кода является ситуация, когда многие пользователи структуры данных проверяют определенное значение, а затем большинство из них выполняют одни и те же действия. Обнаружив во многих частях кодовой базы одинаковую реакцию на определенное значение, я стараюсь перенести эту реакцию в одно место.

Хорошим механизмом для этого является проектный шаблон частного (особого) случая, когда для некоторого случая создается элемент, в котором собрано все обычное поведение для этого случая. Это позволяет заменить большинство проверок частного случая простыми вызовами.

Сам по себе частный случай может проявиться несколькими способами. Если все, что я делаю с объектом, — это чтение данных, я могу предоставить литеральный объект со всеми значениями, которые мне нужно заполнить. Если мне нужно поведение сложнее, чем возврат простого значения, я могу создать специальный объект с методами для всех распространенных действий. Такой объект может быть возвращен инкапсулирующим классом или вставлен в структуру данных с помощью преобразования.

Распространенное значение, требующее специальной обработки, — нулевое, поэтому этот проектный шаблон часто называют *шаблоном нулевого объекта*. Но используемый при этом подход тот же, что и для любого частного случая; так сказать, шаблон нулевого объекта — частный случай шаблона частного случая.

Техника

Начните со структуры данных (или класса) контейнера, который содержит свойство, являющееся предметом рефакторинга. Клиенты контейнера сравнивают свойство `subject` контейнера со значением частного случая. Мы хотим заменить значение частного случая предмета рефакторинга соответствующим классом или структурой данных.

- Добавьте к предмету рефакторинга свойство проверки частного случая, возвращающее `false`.
- Создайте объект частного случая с единственным свойством проверки частного случая, возвращающим значение `true`.
- Примените рефакторинг *Извлечение функции* (с. 152) к коду сравнения для частного случая. Убедитесь, что все клиенты используют новую функцию, а не выполняют сравнение непосредственно.
- Введите в код новый субъект для частного случая, либо возвращая его из вызова функции, либо применяя функцию преобразования.
- Измените тело функции сравнения для частного случая таким образом, чтобы в нем использовалось свойство проверки частного случая.
- Выполните тестирование.
- Воспользуйтесь рефакторингами *Объединение функций в класс* (с. 190) и *Объединение функций в преобразование* (с. 195) для переноса общего поведения частного случая в новый элемент.

  Поскольку класс частного случая обычно возвращает в ответ на простые запросы фиксированные значения, они могут быть обработаны путем предоставления литеральной записи для частного случая.
- Примените рефакторинг *Встраивание функции* (с. 161) к функции сравнения для частного случая в тех местах, где это требуется.

Пример

Коммунальное предприятие предоставляет услуги участкам (домам и квартирам), у которых есть пользователи (customer).

```
class Site...
get customer() {return this._customer;}
```

Пользователь обладает рядом свойств. Я приведу три из них.

```
class Customer...
get name()           {...}
get billingPlan()    {...}
set billingPlan(arg) {...}
get paymentHistory() {...}
```

В большинстве случаев у участка есть пользователь, но бывают ситуации, когда его нет. Кто-то мог уехать, а о въехавшем сведений пока нет. Раз такое может произойти, следует гарантировать обработку нулевого значения любым кодом, который использует объект `customer`. Приведу несколько примеров.

```
Клиент 1...
const aCustomer = site.customer;
// Большое количество промежуточного кода
let customerName;
if (aCustomer === "unknown") customerName = "occupant";
else customerName = aCustomer.name;

Клиент 2...
const plan = (aCustomer === "unknown") ?
        registry.billingPlans.basic
        : aCustomer.billingPlan;

Клиент 3...
if (aCustomer !== "unknown") aCustomer.billingPlan = newPlan;

Клиент 4...
const weeksDelinquent = (aCustomer === "unknown") ?
        0
        : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

Просматривая кодовую базу, я сталкиваюсь со множеством клиентов объекта `site`, которые должны работать с неизвестным пользователем. Большинство из них при обнаружении такового выполняют одинаковые действия: в качестве имени пользователя используют строку `occupant` (обитатель), предоставляют базовый тарифный план и классифицируют его как пользователя с нулевой просрочкой платежей. Часто встречающаяся проверка частного случая плюс общая одинаковая реакция — все это говорит о том, что пришло время для рефакторинга *Введение частного случая*.

Я начинаю с добавления к `Customer` метода, указывающего его неизвестность.

```
class Customer...
get isUnknown() {return false;}
```

Затем добавляю класс неизвестного пользователя.

```
class UnknownCustomer {
  get isUnknown() {return true;}
}
```

Обратите внимание: я не делаю `UnknownCustomer` подклассом класса `Customer`. В других языках, особенно в статически типизированных, я бы поступил именно так, но в JavaScript в силу его правил создания подклассов, а также его динамической типизации лучше так не делать.

Существует один нюанс. Необходимо возвращать этот новый объект частного случая везде, где ожидается `"unknown"`, и заменять каждую проверку неизвестного значения вызовом нового метода `isUnknown`. Вообще говоря, я всегда предпочитаю иметь возможность вносить небольшие изменения по одному, выполняя тестирование. Но если я изменю класс пользователя таким образом, чтобы он возвращал неизвестного пользователя, а не `"unknown"`, я должен сделать так, чтобы каждый пользователь, выполняющий проверку на `"unknown"`, вызывал `isUnknown` — и я вынужден делать все это сразу. Все это мне очень не нравится.

Существует распространенная методика, которую можно использовать в такой ситуации. Я использую рефакторинг *Извлечение функции* (с. 152) с кодом, который мне нужно изменить во многих местах — в данном случае это код сравнения в частном случае.

```
function isUnknown(arg) {
  if (!((arg instanceof Customer) || (arg === "unknown")))
    throw new Error(`investigate bad value: <${arg}>`);
  return (arg === "unknown");
}
```

Я поставил здесь "ловушку" для неожиданных значений. Это может помочь мне обнаружить ошибки или странное поведение при проведении рефакторинга.

Теперь я могу использовать эту функцию всякий раз, когда выполняю проверку неизвестного пользователя. Я могу заменять эти вызовы по одному, выполняя тестирование после каждого изменения.

```
Клиент 1...
let customerName;
if (isUnknown(aCustomer)) customerName = "occupant";
else customerName = aCustomer.name;
```

Через некоторое время я заменю все проверки.

```
Клиент 2...
const plan = (isUnknown(aCustomer)) ?
        registry.billingPlans.basic
        : aCustomer.billingPlan;
```

```
Клиент 3...
if (!isUnknown(aCustomer)) aCustomer.billingPlan = newPlan;

Клиент 4...
const weeksDelinquent = isUnknown(aCustomer) ?
        0
        : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

Изменив все проверки с использованием `isUnknown`, я могу изменить класс `Site` так, чтобы он возвращал неизвестного пользователя.

```
class Site...
get customer() {
  return (this._customer === "unknown") ?
        new UnknownCustomer() : this._customer;
}
```

Я могу проверить, что больше не использую строку `"unknown"`, несколько изменив `isUnknown`.

```
Клиент 1...
function isUnknown(arg) {
  if (!(arg instanceof Customer || arg instanceof UnknownCustomer))
    throw new Error(`investigate bad value: <${arg}>`);
  return arg.isUnknown;
}
```

Я выполняю тестирование, чтобы убедиться, что все нормально работает.

Теперь начинается самое интересное. Я могу использовать рефакторинг *Объединение функций в класс* (с. 190), чтобы взять проверку частного случая каждого конкретного клиента и посмотреть, не могу ли я заменить ее обычно ожидаемым значением. В данный момент у меня есть различные клиенты, использующие слово `"occupant"` в качестве имени неизвестного пользователя, например:

```
Клиент 1...

let customerName;
if (isUnknown(aCustomer)) customerName = "occupant";
else customerName = aCustomer.name;
```

Я добавляю подходящий метод в класс неизвестного пользователя.

```
class UnknownCustomer...
get name() {return "occupant";}
```

Теперь я могу убрать весь условный код.

```
Клиент 1...

const customerName = aCustomer.name;
```

Убедившись, что все нормально работает, я, вероятно, могу также использовать для этой переменной рефакторинг *Встраивание переменной* (с. 169).

Далее идет свойство тарифного плана.

```
Клиент 2...

const plan = (isUnknown(aCustomer)) ?
        registry.billingPlans.basic
        : aCustomer.billingPlan;

Клиент 3...

if (!isUnknown(aCustomer)) aCustomer.billingPlan = newPlan;
```

Для поведения чтения я делаю то же самое, что делал с именем, — использую общий ответ. Что касается поведения записи, то текущий код не вызывает метод установки значения для неизвестного пользователя. Поэтому для частного случая я позволяю вызывать метод установки значения, но он ничего не делает.

```
class UnknownCustomer...
get billingPlan() {return registry.billingPlans.basic;}
set billingPlan(arg) { /* Игнорируем */ }

Чтение клиентом...
const plan = aCustomer.billingPlan;

Запись клиентом...
aCustomer.billingPlan = newPlan;
```

Объекты частного случая являются объектами-значениями и, следовательно, всегда должны быть неизменяемыми, даже если объекты, которые они заменяют, таковыми не являются.

Последний случай немного сложнее, потому что здесь частный случай должен возвращать другой объект, который имеет свои собственные свойства.

```
Клиент...
const weeksDelinquent = isUnknown(aCustomer) ?
        0
        : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

Общее правило для объекта частного случая состоит в том, что если ему необходимо вернуть связанные объекты, то обычно они сами по себе являются частными случаями. Поэтому следует создать нулевую историю платежей.

```
class UnknownCustomer...
get paymentHistory() {return new NullPaymentHistory();}

class NullPaymentHistory...
get weeksDelinquentInLastYear() {return 0;}
```

```
Клиент...
const weeksDelinquent =
    aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

Я продолжаю работу, просматривая всех клиентов, чтобы увидеть, не могу ли я заменить их полиморфным поведением. Но тут будут и исключения, т.е. клиенты, которые в частном случае делают нечто иное. У меня может быть 23 клиента, которые используют в качестве имени неизвестного клиента слово "occupant", но обязательно найдется один "с левой резьбой", которому нужно что-то иное.

```
Клиент...
const name = ! isUnknown(aCustomer) ?
        aCustomer.name
        : "unknown occupant";
```

В этом случае мне нужно сохранить проверку частного случая. Я изменю код таким образом, чтобы использовать метод клиента, по существу, применяя рефакторинг *Встраивание функции* (с. 161) к isUnknown.

```
Клиент...
const name = aCustomer.isUnknown ?
             "unknown occupant"
             : aCustomer.name;
```

По окончании работы со всеми клиентами я могу применить рефакторинг *Удаление неработающего кода* (с. 283) к глобальной функции isPresent, так как больше она никем не должна вызываться.

Пример: использование литерала объекта

Создание класса наподобие рассмотренного — довольно трудоемкая работа для того, что на самом деле является простым значением. В приведенном выше примере я должен был создавать класс, так как пользователь мог быть обновлен. Однако, если в программе имеются только лишь чтения структуры данных, вместо класса можно использовать литеральный объект.

Рассмотрим снова тот же пример — с тем отличием, что на этот раз нет клиента, который обновляет объект пользователя.

```
class Site...
get customer() {return this._customer;}

class Customer...
get name()           {...}
get billingPlan()    {...}
set billingPlan(arg) {...}
get paymentHistory() {...}
```

```
Клиент 1...
const aCustomer = site.customer;
// Большое количество промежуточного кода
let customerName;
if (aCustomer === "unknown") customerName = "occupant";
else customerName = aCustomer.name;

Клиент 2...
const plan = (aCustomer === "unknown") ?
        registry.billingPlans.basic
        : aCustomer.billingPlan;

Клиент 3...
const weeksDelinquent = (aCustomer === "unknown") ?
        0
        : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

Как и в предыдущем примере, я начинаю с добавления свойства `isUnknown` к пользователю и с создания объекта частного случая с этим полем. Отличие только в том, что на этот раз частный случай представляет собой литерал.

```
class Customer...
get isUnknown() {return false;}

Верхний уровень...
function createUnknownCustomer() {
  return {
    isUnknown: true,
  };
}
```

Затем я применяю рефакторинг *Извлечение функции* (с. 152) к проверке условия частного случая.

```
function isUnknown(arg) {
  return (arg === "unknown");
}

Клиент 1...
  let customerName;
  if (isUnknown(aCustomer)) customerName = "occupant";
  else customerName = aCustomer.name;

Клиент 2...
const plan = isUnknown(aCustomer) ?
        registry.billingPlans.basic
        : aCustomer.billingPlan;

Клиент 3...
const weeksDelinquent = isUnknown(aCustomer) ?
        0
        : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

Теперь я изменяю класс `Site` и проверку условия для работы в частном случае.

```
class Site...
get customer() {
  return (this._customer === "unknown") ?
        createUnknownCustomer()
        : this._customer;
}

Верхний уровень...
function isUnknown(arg) {
    return arg.isUnknown;
}
```

Затем заменяю каждый стандартный ответ соответствующим литеральным значением. Начину с имени.

```
function createUnknownCustomer() {
  return {
    isUnknown: true,
    name: "occupant",
  };
}

Клиент 1...
const customerName = aCustomer.name;
```

Затем наступает очередь тарифного плана.

```
function createUnknownCustomer() {
  return {
    isUnknown: true,
    name: "occupant",
    billingPlan: registry.billingPlans.basic,
  };
}

Клиент 2...
const plan = aCustomer.billingPlan;
```

Точно так же с помощью литерала создается вложенная нулевая история платежей.

```
function createUnknownCustomer() {
  return {
    isUnknown: true,
    name: "occupant",
    billingPlan: registry.billingPlans.basic,
    paymentHistory: {
      weeksDelinquentInLastYear: 0,
    },
  };
}
```

```
Клиент 3...
const weeksDelinquent =
    aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

Используя такой литерал, следует сделать его неизменяемым, чего можно достичь с помощью `freeze`. Обычно я предпочитаю использовать класс.

Пример: использование преобразования

В обоих рассмотренных примерах используется класс, но ту же идею можно применить и к записи — с помощью шага преобразования.

Предположим, что наш ввод представляет собой простую структуру записи, которая выглядит примерно следующим образом.

```
{
  name: "Acme Boston",
  location: "Malden MA",
  // Дополнительная информация
  customer: {
    name: "Acme Industries",
    billingPlan: "plan-451",
    paymentHistory: {
      weeksDelinquentInLastYear: 7
    // Дополнительная информация
    },
    // Дополнительная информация
  }
}
```

В некоторых случаях пользователь неизвестен; все такие случаи помечаются одинаково.

```
{
  name: "Warehouse Unit 15",
  location: "Malden MA",
  // Дополнительная информация
  customer: "unknown",
}
```

У меня есть аналогичный код клиента, который выполняет проверку неизвестного пользователя.

```
Клиент 1...
const site = acquireSiteData();
const aCustomer = site.customer;
// Большое количество промежуточного кода
let customerName;
if (aCustomer === "unknown") customerName = "occupant";
else customerName = aCustomer.name;
```

```
Клиент 2...
const plan = (aCustomer === "unknown") ?
        registry.billingPlans.basic
        : aCustomer.billingPlan;

Клиент 3...
const weeksDelinquent = (aCustomer === "unknown") ?
        0
        : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

Вначале следует пропустить структуру данных через преобразование, которое в настоящий момент ограничивается созданием глубокой копии.

```
Клиент 1...
  const rawSite = acquireSiteData();
  const site = enrichSite(rawSite);
  const aCustomer = site.customer;
  // Большое количество промежуточного кода
  let customerName;
  if (aCustomer === "unknown") customerName = "occupant";
  else customerName = aCustomer.name;

function enrichSite(inputSite) {
  return _.cloneDeep(inputSite);
}
```

Я применяю рефакторинг *Извлечение функции* (с. 152) к проверке неизвестного пользователя.

```
function isUnknown(aCustomer) {
  return aCustomer === "unknown";
}

Клиент 1...
const rawSite = acquireSiteData();
const site = enrichSite(rawSite);
const aCustomer = site.customer;
// Большое количество промежуточного кода
let customerName;
if (isUnknown(aCustomer)) customerName = "occupant";
else customerName = aCustomer.name;

Клиент 2...
const plan = (isUnknown(aCustomer)) ?
        registry.billingPlans.basic
        : aCustomer.billingPlan;

Клиент 3...
const weeksDelinquent = (isUnknown(aCustomer)) ?
        0
        : aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

Приступаю к обогащению, добавляя пользователю свойство `isUnknown`.

```
function enrichSite(aSite) {
  const result = _.cloneDeep(aSite);
  const unknownCustomer = {
    isUnknown: true,
  };

  if (isUnknown(result.customer)) result.customer = unknownCustomer;
  else result.customer.isUnknown = false;
  return result;
}
```

Затем можно изменить проверку условия частного случая так, чтобы включить проверку этого нового свойства. Я сохраняю также и исходный тест, чтобы проверка работала как для необработанных, так и для "обогащенных" объектов.

```
function isUnknown(aCustomer) {
  if (aCustomer === "unknown") return true;
  else return aCustomer.isUnknown;
}
```

Как обычно, выполняется тестирование, чтобы убедиться, что все в порядке, а затем — рефакторинг *Объединение функций в преобразование* (с. 195) для частного случая. Сначала я переношу выбор имени в функцию обогащения.

```
function enrichSite(aSite) {
  const result = _.cloneDeep(aSite);
  const unknownCustomer = {
    isUnknown: true,
    name: "occupant",
  };

  if (isUnknown(result.customer))
    result.customer = unknownCustomer;
  else result.customer.isUnknown = false;
  return result;
}


Клиент 1...
const rawSite = acquireSiteData();
const site = enrichSite(rawSite);
const aCustomer = site.customer;
// Большое количество промежуточного кода
const customerName = aCustomer.name;
```

Завершив тестирование, приступаю к работе над тарифным планом.

```
function enrichSite(aSite) {
  const result = _.cloneDeep(aSite);
  const unknownCustomer = {
```

```
    isUnknown: true,
    name: "occupant",
    billingPlan: registry.billingPlans.basic,
  };
  if (isUnknown(result.customer))
    result.customer = unknownCustomer;
  else result.customer.isUnknown = false;
  return result;
}
```

Клиент 2...

```
const plan = aCustomer.billingPlan;
```

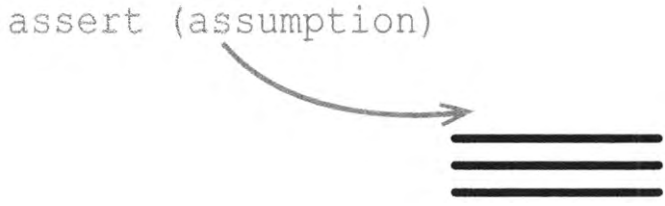
Еще раз выполняю тестирование, после чего обращаюсь к последнему клиенту.

```
function enrichSite(aSite) {
  const result = _.cloneDeep(aSite);
  const unknownCustomer = {
    isUnknown: true,
    name: "occupant",
    billingPlan: registry.billingPlans.basic,
    paymentHistory: {
      weeksDelinquentInLastYear: 0,
    }
  };
  if (isUnknown(result.customer))
    result.customer = unknownCustomer;
  else result.customer.isUnknown = false;
  return result;
}
```

Клиент 3...

```
const weeksDelinquent =
    aCustomer.paymentHistory.weeksDelinquentInLastYear;
```

Введение утверждения (Introduce Assertion)



```
if (this.discountRate)
  base = base - (this.discountRate * base);
```

```
assert(this.discountRate >= 0);
if (this.discountRate)
  base = base - (this.discountRate * base);
```

Мотивация

Часто некоторые разделы кода работают только при выполнении определенных условий. Примером может служить простейшее вычисление квадратного корня, работающее только с положительным аргументом. В случае объекта может потребоваться, чтобы имело значение хотя бы одно из группы его полей.

Такие предположения часто явно не указываются и могут быть обнаружены только при тщательном просмотре алгоритма. Иногда предположения указываются с помощью комментариев. Но лучший способ сделать предположение явным — написать утверждение.

Утверждение (assertion) представляет собой условную конструкцию, которая всегда должна выполняться. Сбой утверждения свидетельствует об ошибке в программе, а потому он должен всегда приводить к необрабатываемому исключению. Утверждения никогда не должны использоваться другими частями системы. В действительности утверждения обычно полностью удаляются из готового кода. Поэтому важно указать, что нечто представляет собой утверждение.

Утверждения способствуют организации сообщений и отладки. Их коммуникативная роль состоит в том, что они помогают читателю понять, какие предположения делает код. Во время отладки утверждения помогают перехватывать ошибки ближе к моменту их возникновения. По моим наблюдениям, при написании самотестирующегося кода помощь утверждений для отладки невелика, но я очень ценю коммуникативную роль утверждений.

Техника

- Когда вы видите, что некоторое условие предполагается всегда выполненным, добавьте утверждение, в котором это сформулировано явно.

Поскольку утверждения не должны оказывать влияния на работу системы, их добавление всегда сохраняет поведение.

Пример

Вот простой рассказ о скидках. Клиенту может быть предоставлена скидка для всех его покупок.

```
class Customer...
applyDiscount(aNumber) {
  return (this.discountRate)
        ? aNumber - (this.discountRate * aNumber)
        : aNumber;
}
```

Здесь предполагается, что дисконтная ставка является положительным числом. Я могу сделать это предположение явным, используя утверждение. Но так как я не могу легко поместить утверждение в тернарное выражение, сначала я переформулирую его как утверждение `if-then`.

```
class Customer...
applyDiscount(aNumber) {
  if (!this.discountRate) return aNumber;
  else return aNumber - (this.discountRate * aNumber);
}
```

Теперь я могу легко добавить утверждение.

```
class Customer...
applyDiscount(aNumber) {
  if (!this.discountRate) return aNumber;
  else {
    assert(this.discountRate >= 0);
    return aNumber - (this.discountRate * aNumber);
  }
}
```

В этом случае я бы предпочел поместить данное утверждение в метод установки значения. Если утверждение в функции `applyDiscount` не выполняется, мне в первую очередь надо узнать, как такое значение сумело попасть в поле `discountRate`.

```
class Customer...
set discountRate(aNumber) {
  assert(null === aNumber || aNumber >= 0);
  this._discountRate = aNumber;
}
```

Подобное утверждение может быть особенно полезным, если трудно определить источник ошибки, который может оказаться ошибочным знаком минус в каких-то входных данных или изменением знака в другом месте кода.

Существует реальная опасность злоупотребления утверждениями. Я не использую утверждения для проверки всего, что считаю истиной, а только для

проверки того, что *обязано* быть правдой. Особую проблему представляет собой дублирование. Поэтому я считаю, что в этих условиях необходимо устранить любое дублирование, обычно путем использования рефакторинга *Извлечение функции* (с. 152).

Я использую утверждения только для проверки ошибок программиста. Если я читаю данные из внешнего источника, любая проверка значений должна быть рабочей частью программы, а не утверждением (если только я не уверен абсолютно в надежности внешнего источника). Утверждения являются последним средством для отслеживания ошибок (хотя, по иронии судьбы, я использую их только тогда, когда рассчитываю, что они всегда будут выполняться и что никаких ошибок в коде нет).

Глава 11

Рефакторинг API

Строительными блоками программного обеспечения являются модули и их функции. Интерфейс прикладного программирования (application programming interface — API) представляет собой соединения, которые используются для построения программ из этих блоков. Важно сделать интерфейсы API простыми для понимания и использования, но это достаточно сложная задача. Поэтому, поняв, как можно их улучшить, следует выполнить их рефакторинг.

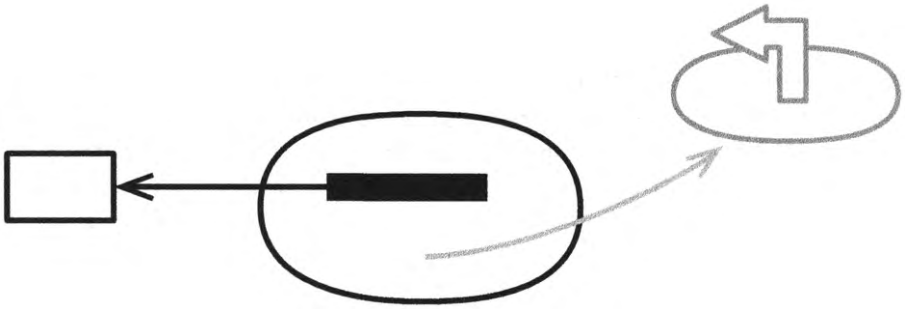
Хороший API четко отделяет любые функции, которые обновляют данные, от функций, которые только читают их. Обнаружив эти функции вместе, я использую рефакторинг *Отделение запроса от модификатора* (с. 352) для их разделения. Я могу объединить функции, которые варьируются только в зависимости от некоторого значения, с помощью рефакторинга *Параметризация функции* (с. 355). Однако некоторые параметры в действительности являются сигналом совершенно другого поведения, и тогда лучше всего прибегнуть к рефакторингу *Удаление аргумента-флага* (с. 359).

Зачастую при передаче между функциями выполняются ненужные распаковки структур данных. В таких случаях я предпочитаю воспользоваться рефакторингом *Сохранение всего объекта* (с. 364). Решения о том, что должно быть передано в качестве параметра, а что может быть разрешено вызываемой функцией, приходится часто пересматривать с использованием рефакторингов *Замена параметра запросом* (с. 369) и *Замена запроса параметром* (с. 372).

Распространенной формой модуля является класс. Я предпочитаю, чтобы мои объекты были по возможности неизменяемыми, и поэтому при любой возможности использую рефакторинг *Удаление метода установки значения* (с. 376). Часто, когда вызывающая сторона запрашивает новый объект, мне требуется большая гибкость, чем может дать простой конструктор, — и в таком случае я пытаюсь получить ее с помощью рефакторинга *Замена конструктора фабричной функцией* (с. 379).

Последние два рефакторинга касаются сложности разбиения особенно сложной функции, которая получает большое количество данных. Я могу превратить эту функцию в объект с помощью рефакторинга *Замена функции командой* (с. 381), что упрощает применение рефакторинга *Извлечение функции* (с. 152) к телу функции. Если позже я упрощу функцию и больше не буду нуждаться в ней как в командном объекте, то снова превращу ее в функцию с помощью рефакторинга *Замена команды функцией* (с. 388).

Отделение запроса от модификатора (Separate Query from Modifier)



```
function getTotalOutstandingAndSendBill() {
  const result =
        customer.invoices.reduce((total, each) =>
        each.amount + total, 0);
  sendBill();
  return result;
}
```

```
function totalOutstanding() {
  return customer.invoices.reduce(
            (total, each) => each.amount + total, 0);
}

function sendBill() {
  emailGateway.send(formatBill(customer));
}
```

Мотивация

Функция, возвращающая значение и не имеющая видимых побочных действий, — весьма ценная вещь: я могу вызывать ее так часто, как захочу, могу перемещать ее вызов в другие места вызывающей функции, а еще ее легче тестировать. Словом, с такой функцией гораздо меньше поводов для беспокойства.

Хорошей идеей является четкое разграничение функций с побочными действиями и без таковых. Правило, которому крайне желательно следовать, заключается в том, что любая функция, которая возвращает значение, не должна иметь никаких наблюдаемых побочных действий — это правило разделения команд и запросов [20]. Некоторые программисты считают это правило абсолютом. Я не

настолько пурист, чтобы безоговорочно настаивать на выполнении этого правила (как и других), но стараюсь следовать ему бо́льшую часть времени (и это сослужило мне хорошую службу).

Сталкиваясь с методом, который возвращает значение, но при этом имеет побочные действия, я всегда пытаюсь отделить запрос от модификатора.

Обратите внимание: я использую фразу "*наблюдаемых* побочных действий". Распространенной оптимизацией является кеширование значения запроса в поле, чтобы повторяющиеся вызовы выполнялись быстрее. Хотя состояние объекта с кешем при этом изменяется — это изменение не является наблюдаемым. Любая последовательность запросов всегда будет возвращать одинаковые результаты для каждого запроса.

Техника

- Скопируйте функцию, дайте ей имя, как функции-запросу.

  Рассмотрите функцию, чтобы понять, что она возвращает. Если запрос используется для заполнения переменной, то хорошим ключом к пониманию должно быть имя переменной.
- Удалите все побочные действия из новой функции-запроса.
- Выполните статические проверки.
- Найдите все вызовы исходной функции. Если вызов использует возвращаемое значение, замените исходный вызов запросом и вставьте вызов исходной функции под ним. Выполните тестирование после каждого изменения.
- Уберите возврат значений из исходной функции.
- Выполните тестирование.

Часто после выполнения этих действий между запросом и исходным методом возникает дублирование, которое можно удалить.

Пример

Вот функция, которая сканирует список имен в поисках злоумышленника. Если она находит его, то возвращает имя плохого парня и включает сигнал тревоги. Она делает это только для первого найденного злоумышленника (думаю, достаточно и одного).

```
function alertForMiscreant(people) {
  for (const p of people) {
    if (p === "Don") {
      setOffAlarms();
      return "Don";
    }
```

```
    if (p === "John") {
      setOffAlarms();
      return "John";
    }
  }
  return "";
}
```

Начну с копирования функции с соответствующим ее именованием, отражающим аспект запроса.

```
function findMiscreant(people) {
  for (const p of people) {
    if (p === "Don") {
      setOffAlarms();
      return "Don";
    }
    if (p === "John") {
      setOffAlarms();
      return "John";
    }
  }
  return "";
}
```

Из этого нового запроса нужно убрать все побочные действия.

```
function findMiscreant(people) {
  for (const p of people) {
    if (p === "Don") {
      ~~setOffAlarms();~~
      return "Don";
    }
    if (p === "John") {
      ~~setOffAlarms();~~
      return "John";
    }
  }
  return "";
}
```

Теперь в каждом месте вызова исходной функции и я заменяю ее вызовом запроса, за которым следует вызов модификатора. Так что

```
const found = alertForMiscreant(people);
```

превращается в

```
const found = findMiscreant(people);
alertForMiscreant(people);
```

Теперь убираю возвращаемое значение убирается из функции-модификатора.

```
function alertForMiscreant (people) {
  for (const p of people) {
    if (p === "Don") {
      setOffAlarms();
      return;
    }
    if (p === "John") {
      setOffAlarms();
      return;
    }
  }
  return;
}
```

У меня оказывается большое количество дублируемого кода между исходным модификатором и новым запросом, так что я могу прибегнуть к рефакторингу *Подстановка алгоритма* (с. 240), чтобы модификатор использовал запрос.

```
function alertForMiscreant (people) {
  if (findMiscreant(people) !== "") setOffAlarms();
}
```

Параметризация функции (Parameterize Function)

Бывший рефакторинг *Параметризация метода*


```
function tenPercentRaise(aPerson) {
  aPerson.salary = aPerson.salary.multiply(1.1);
}
function fivePercentRaise(aPerson) {
  aPerson.salary = aPerson.salary.multiply(1.05);
}
```

```
function raise(aPerson, factor) {
  aPerson.salary = aPerson.salary.multiply(1 + factor);
}
```

Мотивация

Обнаружив две функции со схожей логикой, но с разными литеральными значениями, я могу удалить дублирование, используя одну функцию с параметрами для разных значений. Это увеличивает полезность функции, так как я смогу применять ее в разных местах с другими значениями.

Техника

- Выберите один из схожих методов.
- Примените рефакторинг *Изменение объявления функции* (с. 170) для добавления литералов, которые следует сделать параметрами.
- Для каждого вызова функции добавьте литеральное значение.
- Выполните тестирование.
- Измените тело функции так, чтобы использовать новые параметры. Выполните тестирование после каждого изменения.
- Для каждой подобной функции замените ее вызов вызовом параметризованной функции. Выполните тестирование после каждой замены.

Если исходная параметризованная функция некорректно работает вместо схожей исходной функции, исправьте новую функцию, прежде чем переходить к следующему вызову.

Пример

Очевидный пример имеет такой вид.

```
function tenPercentRaise(aPerson) {
  aPerson.salary = aPerson.salary.multiply(1.1);
}
function fivePercentRaise(aPerson) {
  aPerson.salary = aPerson.salary.multiply(1.05);
}
```

Очевидно, что я могу заменить эти функции следующей.

```
function raise(aPerson, factor) {
  aPerson.salary = aPerson.salary.multiply(1 + factor);
}
```

Но ситуация может оказаться и немного более сложной. Рассмотрим такой код.

```
function baseCharge(usage) {
  if (usage < 0) return usd(0);
  const amount =
            bottomBand(usage) * 0.03
```

```
          + middleBand(usage) * 0.05
          + topBand(usage) * 0.07;
  return usd(amount);
}

function bottomBand(usage) {
  return Math.min(usage, 100);
}

function middleBand(usage) {
  return usage > 100 ? Math.min(usage, 200) - 100 : 0;
}

function topBand(usage) {
  return usage > 200 ? usage - 200 : 0;
}
```

Здесь логика явно очень похожа, но в достаточной ли мере, чтобы поддержать создание параметризованного метода? Да, это так, но выглядит этот случай куда менее очевидным, чем приведенный выше тривиальный код.

При поиске способа параметризации некоторых связанных функций мой подход заключается в том, чтобы взять одну из функций и добавлять к ней параметры с учетом других случаев. В такой ситуации, как здесь, — ориентированной на диапазон, — обычно лучше начинать со средины диапазона. Поэтому я начинаю работу над `middleBand`, чтобы позволить этой функции использовать параметры, а затем я выполню ее настройку под свои нужды.

`middleBand` использует два литеральных значения: `100` и `200`. Начну с применения рефакторинга *Изменение объявления функции* (с. 170), чтобы добавить их в вызов. Пока я занимаюсь этой работой, заодно изменю имя функции на иное, имеющее смысл при использовании параметризации.

```
function withinBand(usage, bottom, top) {
  return usage > 100 ? Math.min(usage, 200) - 100 : 0;
}

function baseCharge(usage) {
  if (usage < 0) return usd(0);
  const amount =
            bottomBand(usage) * 0.03
          + withinBand(usage, 100, 200) * 0.05
          + topBand(usage) * 0.07;
  return usd(amount);
}
```

Заменяю каждый литерал обращением к параметру:

```
function withinBand(usage, bottom, top) {
  return usage > bottom ? Math.min(usage, 200) - bottom : 0;
}
```

затем:

```
function withinBand(usage, bottom, top) {
  return usage > bottom ? Math.min(usage, top) - bottom : 0;
}
```

Далее я заменяю вызов `bottomBand` вызовом новой параметризованной функции.

```
function baseCharge(usage) {
  if (usage < 0) return usd(0);
  const amount =
        withinBand(usage, 0, 100) * 0.03
      + withinBand(usage, 100, 200) * 0.05
      + topBand(usage) * 0.07;
  return usd(amount);
}

function bottomBand(usage) {
  return Math.min(usage, 100);
}
```

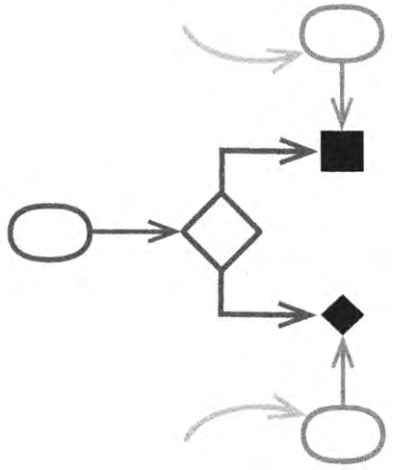
Для замены `topBand` мне нужно использовать бесконечность.

```
function baseCharge(usage) {
  if (usage < 0) return usd(0);
  const amount =
        withinBand(usage, 0, 100) * 0.03
      + withinBand(usage, 100, 200) * 0.05
      + withinBand(usage, 200, Infinity) * 0.07;
  return usd(amount);
}

function topBand(usage) {
  return usage > 200 ? usage - 200 : 0;
}
```

При нынешней логике я мог бы убрать начальную проверку. Но, хотя сейчас она и не нужна с точки зрения логики, я предпочел сохранить ее, так как она представляет собой документацию о том, как следует обрабатывать этот случай.

Удаление аргумента-флага (Remove Flag Argument)

Бывший рефакторинг *Замена параметра явными методами*



```
function setDimension(name, value) {
  if (name === "height") {
    this._height = value;
    return;
  }
  if (name === "width") {
    this._width = value;
    return;
  }
}
```

```
function setHeight(value) {this._height = value;}
function setWidth (value) {this._width = value;}
```

Мотивация

Аргумент-флаг — это аргумент функции, который вызывающая функция использует, чтобы указать, какую именно логику должна выполнять вызываемая функция. Я могу вызвать функцию, которая выглядит следующим образом.

```
function bookConcert(aCustomer, isPremium) {
  if (isPremium) {
    // Логика для премиум-класса
  } else {
    // Обычная логика
  }
}
```

Чтобы выполнить премиум-заказ, функция вызывается следующим образом:

```
bookConcert(aCustomer, true);
```

Аргументы-флаги могут быть и перечислениями:

```
bookConcert(aCustomer, CustomerType.PREMIUM);
```

и строками (или символами использующего их языка):

```
bookConcert(aCustomer, "premium");
```

Мне не нравятся аргументы-флаги, потому что они усложняют процесс понимания того, какие вызовы функций доступны и как именно их вызывать. Мое первое знакомство с API обычно начинается со списка доступных функций, а аргументы-флаги скрывают различия в доступных вызовах функций. Выбрав функцию, я должен еще и выяснить, какие значения аргументов-флагов допустимы. При этом булевы флаги оказываются еще хуже, так как они сами не поясняют свое значение читателю — как при вызове функции я могу понять, что означает `true`? Более понятным решением является предоставление явной функции для каждой решаемой задачи.

```
premiumBookConcert(aCustomer);
```

Не все аргументы, подобные рассматриваемым, являются аргументами-флагами. Чтобы быть аргументом-флагом, вызывающая функция должна установить логическое значение равным литеральному значению, а не брать данные, получаемые в программе. Кроме того, реализуемая функция должна использовать этот аргумент для воздействия на свой поток управления, а не в качестве данных, которые она передает другим функциям.

Удаление аргументов-флагов не только делает код более понятным, но и помогает используемому мною инструментарию. Так, инструментам анализа кода теперь легче увидеть разницу между вызовом логики премиум-класса и вызовом обычной логики.

Аргументы-флаги имеют право на существование, если в функции их несколько, поскольку в противном случае мне потребуются явные функции для каждой комбинации их значений. Но это одновременно является сигналом того, что функция делает слишком многое, и следует искать способ создания более простых функций, которые могут быть скомбинированы для реализации этой сложной логики.

Техника

- Создайте явную функцию для каждого значения параметра.

  Если у главной функции есть четкая диспетчеризация по условию, для создания явных функций используйте рефакторинг *Декомпозиция условной инструкции* (с. 306). В противном случае создайте функции-оболочки.

- Каждый вызов, который в качестве аргумента использует литеральное значение, заменяем вызовом явной функции.

Пример

Просматривая код, я вижу вызовы вычисления даты доставки. Одни вызовы имеют такой вид:

```
aShipment.deliveryDate = deliveryDate(anOrder, true);
```

а другие — такой:

```
aShipment.deliveryDate = deliveryDate(anOrder, false);
```

Столкнувшись с таким кодом, я сразу же начинаю задумываться о том, для чего служит передаваемое в функцию логическое значение.

Тело функции `deliveryDate` имеет следующий вид.

```
function deliveryDate(anOrder, isRush) {
  if (isRush) {
    let deliveryTime;
    if (["MA","CT"].includes(anOrder.deliveryState))
        deliveryTime = 1;
    else if (["NY","NH"].includes(anOrder.deliveryState))
        deliveryTime = 2;
    else
        deliveryTime = 3;
    return anOrder.placedOn.plusDays(1 + deliveryTime);
  }
  else {
    let deliveryTime;
    if (["MA","CT","NY"].includes(anOrder.deliveryState))
        deliveryTime = 2;
    else if (["ME","NH"].includes(anOrder.deliveryState))
        deliveryTime = 3;
    else
        deliveryTime = 4;
    return anOrder.placedOn.plusDays(2 + deliveryTime);
  }
}
```

Здесь литеральное логическое значение используется для того, чтобы определить, какой именно код должен выполняться — т.е. оно представляет собой классический аргумент-флаг. Весь смысл использования функции заключается в том, чтобы следовать инструкциям вызывающей функции, поэтому лучше уточнить намерения вызывающего кода с использованием явных функций.

В данном случае я могу сделать это с помощью рефакторинга *Декомпозиция условной инструкции* (с. 306), который позволяет мне получить следующий код.

```
function deliveryDate(anOrder, isRush) {
  if (isRush) return rushDeliveryDate(anOrder);
  else return regularDeliveryDate(anOrder);
}

function rushDeliveryDate(anOrder) {
  let deliveryTime;
  if (["MA","CT"].includes(anOrder.deliveryState))
    deliveryTime = 1;
  else if (["NY","NH"].includes(anOrder.deliveryState))
    deliveryTime = 2;
  else
    deliveryTime = 3;
  return anOrder.placedOn.plusDays(1 + deliveryTime);
}

function regularDeliveryDate(anOrder) {
  let deliveryTime;
  if (["MA","CT","NY"].includes(anOrder.deliveryState))
    deliveryTime = 2;
  else if (["ME","NH"].includes(anOrder.deliveryState))
    deliveryTime = 3;
  else
    deliveryTime = 4;
  return anOrder.placedOn.plusDays(2 + deliveryTime);
}
```

Эти две новые функции лучше указывают цель их вызова, так что я могу заменить каждый вызов

```
aShipment.deliveryDate = deliveryDate(anOrder, true);
```

вызовом

```
aShipment.deliveryDate = rushDeliveryDate(anOrder);
```

(и сделать то же самое со вторым случаем).

После замены всех вызовов я удаляю `deliveryDate`.

Аргумент-флаг — это не просто наличие логического значения; это задание логического значения с использованием литерала, а не данных. Если бы все вызовы `deliveryDate` имели вид

```
const isRush = determineIfRush(anOrder);
aShipment.deliveryDate = deliveryDate(anOrder, isRush);
```

то у меня не было бы проблем с сигнатурой `deliveryDate` (хотя все равно следовало бы применить рефакторинг *Декомпозиция условной инструкции* (с. 306)).

Может случиться так, что некоторые вызовы используют аргумент как аргумент-флаг, указывая литерал, в то время как другие вызовы передают некоторые данные. В таком случае я по-прежнему воспользуюсь рассматриваемым рефакторингом, но не буду изменять вызовы с передачей данных и не стану в конце работы удалять функцию `deliveryDate`. Таким образом, я поддерживаю оба интерфейса для разных целей.

Декомпозиция условного выражения — хороший способ выполнить рассматриваемый рефакторинг, но она работает только в том случае, если диспетчеризация по параметру является внешней частью функции (или я могу легко выполнить рефакторинг, чтобы сделать ее таковой). Может оказаться, что параметр используется намного более запутанным способом, как, например, в следующей альтернативной версии `deliveryDate`.

```
function deliveryDate(anOrder, isRush) {
  let result;
  let deliveryTime;
  if (anOrder.deliveryState === "MA" ||
      anOrder.deliveryState === "CT")
          deliveryTime = isRush? 1 : 2;
  else if (anOrder.deliveryState === "NY" ||
           anOrder.deliveryState === "NH") {
          deliveryTime = 2;
  if (anOrder.deliveryState === "NH" && !isRush)
          deliveryTime = 3;
  }
  else if (isRush)
      deliveryTime = 3;
  else if (anOrder.deliveryState === "ME")
      deliveryTime = 3;
  else
      deliveryTime = 4;

  result = anOrder.placedOn.plusDays(2 + deliveryTime);
  if (isRush) result = result.minusDays(1);

  return result;
}
```

В этом случае, пожалуй, овчинка не стоит выделки, так что вместо применения указанного рефакторинга я просто создаю функции поверх `deliveryDate`.

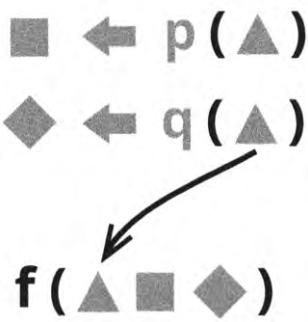
```
function rushDeliveryDate (anOrder) {
    return deliveryDate(anOrder, true);
}
```

```
function regularDeliveryDate(anOrder) {
    return deliveryDate(anOrder, false);
}
```

Эти функции-обертки по сути являются частными случаями применения `deliveryDate`.

Затем я могу выполнить ту же замену вызовов функции, что я делал ранее после декомпозиции условной конструкции. Если никаких вызовов, использующих в качестве аргумента данные, в программе нет, то я предпочел бы ограничить видимость исходной функции или переименовать ее таким образом, чтобы было понятно, что непосредственно использовать ее нельзя (например, дав ей имя наподобие `deliveryDateHelperOnly`).

Сохранение всего объекта (Preserve Whole Object)




```
const low = aRoom.daysTempRange.low;
const high = aRoom.daysTempRange.high;
if (aPlan.withinRange(low, high))
```

```
if (aPlan.withinRange(aRoom.daysTempRange))
```

Мотивация

Если я вижу код, который извлекает из записи пару значений, а затем передает эти значения в функцию, я предпочитаю заменить эти значения на всю запись целиком, позволяя функции самой извлечь нужные ей значения в своем теле.

Передача всей записи в целом в особенности оправдывает себя в случае, если в будущем вызываемой функции потребуется больше данных из целого объекта — в случае передачи всей записи такое изменение не потребует от меня изменения

списка параметров. Кроме того, такое решение уменьшает размер списка параметров, что обычно делает вызов функции более легким для понимания. Если ряду функций передается только частичная информация, получаемая от объекта, то чаще всего это означает дублирование логики, работающей с разными частями объекта, — логики, которую можно было бы перенести в работу с целым объектом.

Если я предпочитаю так не делать — то основная причина такого моего решения заключается в том, что я не хочу зависимости вызываемой функции от целого объекта (обычно я принимаю такое решение, если функция и объект находятся в разных модулях).

Извлечение нескольких значений из объекта для выполнения некоторой логики с ними определенно является запахом (см. раздел "Завистливые функции" главы 3). Как правило, этот запах сигнализирует о том, что логику следует перенести в сам объект в целом. Рассматриваемый рефакторинг обычно применяется после рефакторинга *Введение объекта параметра* (с. 186), так как я отслеживаю все вхождения исходных данных, чтобы заменить их новым объектом.

Если несколько фрагментов кода используют только одно и то же подмножество функциональных возможностей объекта — это может указывать на возможность применения рефакторинга *Извлечение класса* (с. 152).

Есть один случай, который многие склонны не замечать — это когда один объект вызывает другой объект с несколькими значениями собственных данных. В таком случае я стремлюсь заменить эти значения ссылкой на себя (`this` в JavaScript).

Техника

- Создайте пустую функцию с требуемыми параметрами.

  Дайте функции простое для поиска имя, чтобы его можно было легко заменить в конце работы.

- Внесите в тело новой функции вызов старой, отображая новые параметры на старые.
- Выполните статические проверки.
- Обновите каждый вызов так, чтобы он использовал новую функцию. Выполните тестирование после каждого изменения.

  Это может означать, что некоторый код, получающий значение аргумента для старого вызова, становится не нужен, так что можно прибегнуть к рефакторингу *Удаление неработающего кода* (с. 283).

- Заменив вызовы старой функции, примените к исходной функции рефакторинг *Встраивание функции* (с. 161).
- Измените имя новой функции и всех ее вызовов.

Пример

Рассмотрим систему мониторинга помещения. Она сравнивает диапазон дневных температур с диапазоном в предопределенном плане отопления.

```
Точка вызова...
const low = aRoom.daysTempRange.low;
const high = aRoom.daysTempRange.high;
if (!aPlan.withinRange(low, high))
  alerts.push("температура вышла за пределы диапазона");

class HeatingPlan...
withinRange(bottom, top) {
  return (bottom >= this._temperatureRange.low) &&
         (top <= this._temperatureRange.high);
}
```

Вместо распаковки информации о диапазоне при ее передаче в функцию я могу передать объект диапазона в целом.

Начину с описания желательного интерфейса в виде пустой функции.

```
class HeatingPlan...
  xxNEWwithinRange(aNumberRange) {
}
```

Поскольку я намереваюсь заменить существующую функцию `withinRange`, новую функцию я называю тем же именем, но с легко заменяемым префиксом.

Затем добавляю тело функции, основанное на вызове существующей функции `withinRange`. Таким образом, тело функции состоит из отображения нового параметра на существующие.

```
class HeatingPlan...
xxNEWwithinRange(aNumberRange) {
  return this.withinRange(aNumberRange.low,
                          aNumberRange.high);
}
```

Теперь можно начать серьезную работу, находя существующие вызовы функций и заставляя их вызывать новую функцию.

```
Точка вызова...
const low = aRoom.daysTempRange.low;
const high = aRoom.daysTempRange.high;
if (!aPlan.xxNEWwithinRange(aRoom.daysTempRange))
  alerts.push("температура вышла за пределы диапазона");
```

По завершении изменения вызовов становится ясно, что некоторый старый код оказывается больше не нужен, поэтому я использую рефакторинг *Удаление неработающего кода* (с. 283).

```
Точка вызова...
const low = aRoom.daysTempRange.low;
const high = aRoom.daysTempRange.high;
if (!aPlan.xxNEWwithinRange(aRoom.daysTempRange))
  alerts.push("температура вышла за пределы диапазона");
```

Я выполняю замены по одной, с тестированием после каждого внесенного изменения.

Выполнив все необходимые замены, я применяю рефакторинг *Встраивание функции* (с. 161) к исходной функции.

```
class HeatingPlan...
xxNEWwithinRange(aNumberRange) {
  return (aNumberRange.low >= this._temperatureRange.low) &&
         (aNumberRange.high <= this._temperatureRange.high);
}
```

И, наконец, я удаляю этот уродливый префикс из имени новой функции и всех точек ее вызова. Даже если в моем редакторе нет надежной поддержки переименования функций, это делается простой глобальной заменой.

```
class HeatingPlan...
withinRange(aNumberRange) {
  return (aNumberRange.low >= this._temperatureRange.low) &&
         (aNumberRange.high <= this._temperatureRange.high);
}
Точка вызова...
if (!aPlan.withinRange(aRoom.daysTempRange))
  alerts.push("температура вышла за пределы диапазона");
```

Пример: вариация для создания новой функции

В приведенном выше примере код новой функции был написан непосредственно. В большинстве случаев это простой и легкий путь. Но у данного рефакторинга имеется полезная в ряде случаев вариация, которая позволяет создать новую функцию полностью из рефакторингов.

Я начинаю с вызова существующей функции.

```
Точка вызова...
const low = aRoom.daysTempRange.low;
const high = aRoom.daysTempRange.high;
if (!aPlan.withinRange(low, high))
  alerts.push("температура вышла за пределы диапазона");
```

Я хочу преобразовать код так, чтобы иметь возможность создать новую функцию с помощью рефакторинга *Извлечение функции* (с. 152), примененного к некоторому существующему коду. Вызывающего кода пока что недостаточно, но я могу получить нужный мне код, несколько раз применяя рефакторинг *Извлечение*

переменной (с. 165). Сначала я отделю вызов старой функции от условной конструкции.

```
Точка вызова...
const low = aRoom.daysTempRange.low;
const high = aRoom.daysTempRange.high;
const isWithinRange = aPlan.withinRange(low, high);
if (!isWithinRange)
  alerts.push("температура вышла за пределы диапазона");
```

Затем я извлекаю входной параметр.

```
Точка вызова...
const tempRange = aRoom.daysTempRange;
const low = tempRange.low;
const high = tempRange.high;
const isWithinRange = aPlan.withinRange(low, high);
if (!isWithinRange)
  alerts.push("температура вышла за пределы диапазона");
```

Сделав все это, можно прибегнуть к рефакторингу *Извлечение функции* (с. 152) для создания новой функции.

```
Точка вызова...
const tempRange = aRoom.daysTempRange;
const isWithinRange = xxNEWwithinRange(aPlan, tempRange);
if (!isWithinRange)
  alerts.push("температура вышла за пределы диапазона");

Верхний уровень...
function xxNEWwithinRange(aPlan, tempRange) {
  const low = tempRange.low;
  const high = tempRange.high;
  const isWithinRange = aPlan.withinRange(low, high);
  return isWithinRange;
}
```

Поскольку исходная функция находится в другом контексте (класс `HeatingPlan`), я должен использовать рефакторинг *Перенос функции* (с. 244).

```
Точка вызова...
const tempRange = aRoom.daysTempRange;
const isWithinRange = aPlan.xxNEWwithinRange(tempRange);
if (!isWithinRange)
  alerts.push("температура вышла за пределы диапазона");

class HeatingPlan...
xxNEWwithinRange(tempRange) {
  const low = tempRange.low;
  const high = tempRange.high;
  const isWithinRange = this.withinRange(low, high);
  return isWithinRange;
}
```

Затем я продолжаю действовать, как и ранее, заменяя другие вызовы и встраивая старую функцию в новую. Я бы также встроил извлеченные переменные, чтобы обеспечить полное разделение для извлечения новой функции.

Поскольку этот вариант полностью состоит из рефакторингов, он особенно удобен, если у меня есть инструментарий для автоматического рефакторинга с надежными операциями извлечения и встраивания.

Замена параметра запросом (Replace Parameter with Query)

Бывший рефакторинг *Замена параметра вызовом метода*
Обратный к рефакторингу *Замена запроса параметром* (с. 372)


```
availableVacation(anEmployee, anEmployee.grade);

function availableVacation(anEmployee, grade) {
  // Расчет отпуска...
```

```
availableVacation(anEmployee)

function availableVacation(anEmployee) {
  const grade = anEmployee.grade;
  // Расчет отпуска...
```

Мотивация

Список параметров функции подытоживает ее изменчивость, указывая основные пути различного поведения этой функции. Чем меньше список аргументов функции, тем легче понять, как она работает.

Если вызов передает функции значение, которое та может столь же легко определить самостоятельно, — это по сути является формой дублирования, усложняющего вызывающую функцию (которая должна определить значение параметра и передать его, при том, что ее можно полностью освободить от этой работы).

Основное ограничение в предыдущем абзаце выражено фразой "столь же легко". Удаляя параметр, я перекладываю на функцию ответственность за

определение значения этого параметра. При наличии параметра определение его значения — обязанность вызывающей функции; в противном случае эта ответственность переходит к вызываемой функции. Я привык упрощать жизнь вызывающим функциям, что подразумевает перенос ответственности в тело вызываемой функции, — но только если перенос этой ответственности в нее уместен.

Самая распространенная причина, по которой следует избегать применения рассматриваемого рефакторинга, заключается в том, что удаление параметра добавляет нежелательную зависимость в тело функции, заставляя его обращаться к элементу программы, который этой функции знать не обязательно. Это может быть как новая зависимость, так и существующая, которую я хотел бы удалить.

Наиболее безопасный случай для применения рассматриваемого рефакторинга *Замена параметра запросом* — когда параметр, который я хочу удалить, определяется простым запросом другого параметра в списке параметров функции. Редко имеет смысл передача двух параметров, если один из них можно вычислить на основании другого.

Следует обратить внимание на наличие у функции ссылочной прозрачности — т.е. будет ли функция при вызове с одними и теми же значениями параметров вести себя одинаково. С такими функциями гораздо проще работать; их гораздо проще тестировать, и я не хотел бы потерять это свойство функции при ее рефакторинге. Именно поэтому я не стану заменять параметр, например, обращением к изменяемой глобальной переменной.

Техника

- При необходимости примените рефакторинг *Извлечение функции* (с. 152) к вычислению параметра.
- Замените обращения к параметру в теле функции обращением к выражению, возвращающему значение параметра. Выполните тестирование после внесения каждого изменения.
- Воспользуйтесь рефакторингом *Изменение объявления функции* (с. 170) для удаления параметра.

Пример

Чаще всего я прибегаю к рефакторингу *Замена параметра запросом* при выполнении некоторых других рефакторингов, которые делают параметр больше не нужным. Рассмотрим следующий код.

```
class Order...
get finalPrice() {
  const basePrice = this.quantity * this.itemPrice;
  let discountLevel;
```

```
  if (this.quantity > 100) discountLevel = 2;
  else discountLevel = 1;
  return this.discountedPrice(basePrice, discountLevel);
}

discountedPrice(basePrice, discountLevel) {
  switch (discountLevel) {
    case 1: return basePrice * 0.95;
    case 2: return basePrice * 0.9;
  }
}
```

При упрощении функции я стремлюсь использовать рефакторинг *Замена временной переменной запросом* (с. 225), который приводит меня к коду

```
class Order...
get finalPrice() {
  const basePrice = this.quantity * this.itemPrice;
  return this.discountedPrice(basePrice, this.discountLevel);
}

get discountLevel() {
  return (this.quantity > 100) ? 2 : 1;
}
```

Теперь нет необходимости передавать результат `discountLevel` в `discountedPrice`, так как последняя функция сама может столь же легко вызвать первую.

Я заменяю все обращения к параметру вызовом метода.

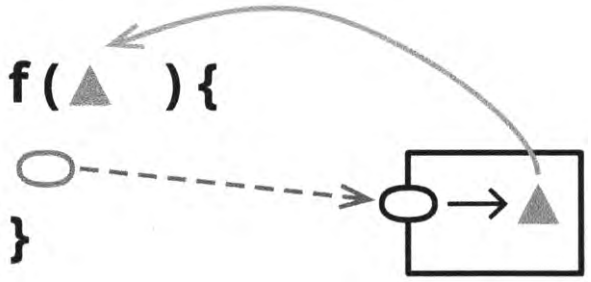
```
class Order...
discountedPrice(basePrice, discountLevel) {
  switch (this.discountLevel) {
    case 1: return basePrice * 0.95;
    case 2: return basePrice * 0.9;
  }
}
```

Затем можно использовать рефакторинг *Изменение объявления функции* (с. 170) и удалить ненужный более параметр.

```
class Order...
get finalPrice() {
  const basePrice = this.quantity * this.itemPrice;
  return this.discountedPrice(basePrice~~, this.discountLevel~~);
}
discountedPrice(basePrice~~, discountLevel~~) {
  switch (this.discountLevel) {
    case 1: return basePrice * 0.95;
    case 2: return basePrice * 0.9;
  }
}
```

Замена запроса параметром (Replace Query with Parameter)

Обратный к рефакторингу *Замена параметра запросом* (с. 369)




```
targetTemperature(aPlan)

function targetTemperature(aPlan) {
  currentTemperature = thermostat.currentTemperature;
  // Остальной код функции...
```

```
targetTemperature(aPlan, thermostat.currentTemperature)

function targetTemperature(aPlan, currentTemperature) {
  // Остальной код функции...
```

Мотивация

Просматривая тело функции, я иногда встречаю в области видимости функции ссылку на нечто, что меня не устраивает. Это может быть ссылка на глобальную переменную или обращение к элементу в том же модуле, который я собираюсь удалить. Чтобы решить эту проблему, мне нужно заменить внутреннюю ссылку параметром, перекладывая ответственность за разрешение ссылки на вызывающую функцию.

Большинство таких случаев связано с моим желанием изменить отношения взаимозависимости в коде, чтобы целевая функция больше не зависела от элемента, который я хочу параметризовать. Однако преобразование различных сущностей в параметры ведет к длинным повторяющимся спискам параметров и совместному использованию областей видимости большого размера, что зачастую приводит к еще большей взаимозависимости между функциями. Оценить ситуацию и принять правильное решение достаточно сложно. Оно требует хорошего

понимания происходящего, чтобы программа могла получить выгоду от такого преобразования.

Проще работать с функцией, которая при вызове с одинаковыми значениями параметров всегда будет давать один и тот же результат. Такое свойство функции называется *ссылочной прозрачностью*. Если функция обращается к некоторому элементу в своей области видимости, который не обладает ссылочной прозрачностью, то и сама функция также не обладает ею. Но такое положение дел можно исправить, переместив непрозрачный элемент в параметр. Несмотря на то что такой шаг перекладывает ответственность на вызывающую функцию, часто можно получить немалую выгоду от создания четко определенных модулей со ссылочной прозрачностью. Обычным является шаблон наличия модулей, состоящих из чистых функций, которые "обернуты" логикой, обрабатывающей ввод-вывод и прочие переменные элементы программы. Я могу использовать рефакторинг *Замена запроса параметром* для соответствующей "очистки" частей программы, облегчающей их тестирование и анализ.

Но было бы ошибкой считать данный рефакторинг обладающим одними лишь преимуществами. Перемещая запрос в параметр, я заставляю вызывающую функцию выводить и предоставлять указанное значение. Это усложняет жизнь вызывающим функциям и их разработчикам, а я обычно склоняюсь к разработке интерфейсов, которые облегчают жизнь программистам. В конечном итоге вопрос сводится к распределению ответственности в пределах программы, а такое решение не может быть ни легким, ни неизменяемым. Вот почему нужно очень хорошо знать данный рефакторинг (и обратный к нему) и понимать все плюсы и минусы их применения в конкретной ситуации.

Техника

- Примените рефакторинг *Извлечение переменной* (с. 165) к коду запроса, чтобы отделить его от остального тела функции.
- Примените рефакторинг *Извлечение функции* (с. 152) к коду тела функции, не являющемуся вызовом запроса.

  Дайте новой функции имя, которое будет легко найти при последующем переименовании.
- Используйте рефакторинг *Встраивание переменной* (с. 169)), чтобы избавиться от только что созданной переменной.
- Примените рефакторинг *Встраивание функции* (с. 161) к исходной функции.
- Переименуйте новую функцию, дав ей имя исходной функции.

Пример

Рассмотрим простую (но уже раздражающую) систему управления температурой. Она позволяет пользователю выбрать температуру на термостате, но устанавливает целевую температуру только в пределах диапазона, определенного планом отопления.

```
class HeatingPlan...
get tgtTemperature() {
  if      (thermostat.selectedTemperature > this._max)
         return this._max;
  else if (thermostat.selectedTemperature < this._min)
         return this._min;
  else   return thermostat.selectedTemperature;
}
```

```
Точка вызова...
if (thePlan.tgtTemperature >
    thermostat.currentTemperature)      setToHeat();
else if (thePlan.tgtTemperature <
         thermostat.currentTemperature) setToCool();
else setOff();
```

Меня, как пользователя такой системы, может раздражать, что правила плана отопления превалируют над моими желаниями, но как программиста меня больше беспокоит тот факт, что функция `tgtTemperature` зависит от объекта глобального термостата. Я могу разорвать эту зависимость, переместив ее в параметр.

Мой первый шаг состоит в применении рефакторинга *Извлечение переменной* (с. 165) к параметру, который я хочу добавить в мою функцию.

```
class HeatingPlan...
get tgtTemperature() {
  const selectedTemperature = thermostat.selectedTemperature;
  if      (selectedTemperature > this._max) return this._max;
  else if (selectedTemperature < this._min) return this._min;
  else return selectedTemperature;
}
```

Это упрощает применение рефакторинга *Извлечение функции* (с. 152) ко всему телу функции, за исключением фрагмента вывода параметра.

```
class HeatingPlan...
get tgtTemperature() {
  const selectedTemperature = thermostat.selectedTemperature;
  return this.xxNEWtgtTemp(selectedTemperature);
}

xxNEWtgtTemp(selectedTemperature) {
  if      (selectedTemperature > this._max) return this._max;
```

```
  else if (selectedTemperature < this._min) return this._min;
  else return selectedTemperature;
}
```

Затем я встраиваю только что извлеченную переменную, что делает функцию простым вызовом.

```
class HeatingPlan...
get tgtTemperature() {
  return this.xxNEWtgtTemp(thermostat.selectedTemperature);
}
```

Теперь я могу применить к этому методу рефакторинг *Встраивание функции* (с. 161).

```
Точка вызова...
if      (thePlan.xxNEWtgtTemp(thermostat.selectedTemperature) >
         thermostat.currentTemperature)
  setToHeat();
else if (thePlan.xxNEWtgtTemp(thermostat.selectedTemperature) <
         thermostat.currentTemperature)
  setToCool();
else
  setOff();
```

Теперь я переименую новую функцию, возвращая ей имя исходной функции.

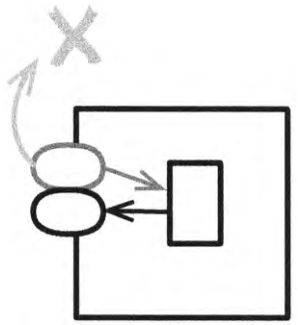
```
Точка вызова...
if       (thePlan.tgtTemperature(thermostat.selectedTemperature)>
         thermostat.currentTemperature)
  setToHeat();
else if (thePlan.tgtTemperature(thermostat.selectedTemperature)<
         thermostat.currentTemperature)
  setToCool();
else
  setOff();

class HeatingPlan...
tgtTemperature(selectedTemperature) {
  if      (selectedTemperature > this._max) return this._max;
  else if (selectedTemperature < this._min) return this._min;
  else return selectedTemperature;
}
```

Как часто бывает при выполнении этого рефакторинга, вызывающий код выглядит более громоздким, чем раньше. Перенос зависимости из модуля передает ответственность за работу с этой зависимостью вызывающей стороне. Это компромиссное решение для уменьшения связанности.

Но удаление связанности с объектом термостата — не единственное преимущество, получаемое с помощью данного рефакторинга. Класс `HeatingPlan` является неизменяемым — его поля устанавливаются в конструкторе, и нет

никаких методов для их изменения. (Я избавлю вас от усилий по изучению всего класса — просто поверьте этому моему заявлению.) С учетом неизменяемости плана отопления удаление ссылки на термостат из тела функции делает функцию `tgtTemperature` ссылочно прозрачной. Каждый раз, вызывая `tgtTemperature` для одного и того же объекта с одним и тем же аргументом, я получаю один и тот же результат. Если все методы плана отопления обладают ссылочной прозрачностью, это значительно упрощает проверку и анализ этого класса. Создание классов, имеющих эту характеристику, зачастую оказывается весьма разумной стратегией, и удобным инструментом для этого является рассматриваемый в данном разделе рефакторинг.

Удаление метода установки значения (Remove Setting Method)



```
class Person {
  get name() {...}
  set name(aString) {...}
```

```
class Person {
  get name() {...}
```

Мотивация

Предоставление метода установки значения поля означает, что это изменяемое поле. Но если я не хочу, чтобы это поле изменялось после создания объекта, я такой метод не предоставлю (и тем самым сделаю поле неизменяемым). Таким образом, значение поля устанавливается только в конструкторе, мое намерение

не изменять его делается очевидным, и я удаляю саму возможность изменения значения поля.

Есть несколько распространенных случаев, когда требуется выполнение данного рефакторинга. Одним из них является ситуация, когда программисты везде используют методы доступа для работы с полем — даже внутри конструкторов. Это приводит к единственному вызову метода установки значения из конструктора. В таком случае я предпочитаю удалить метод установки значения, чтобы ясно показать, что обновления поля после создания объекта не имеют смысла (и попросту невозможны).

Еще один случай — когда объект создается клиентами, использующими сценарий создания, а не простой вызов конструктора. Такой сценарий создания начинается с вызова конструктора, за которым следует последовательность вызовов методов установки значений полей для настройки нового объекта. После завершения такого сценария мы не ожидаем никаких изменений значений некоторых (или даже всех) полей нового объекта. Ожидается, что методы установки значений будут вызываться только во время этого сценария создания. В таком случае я бы постарался избавиться от этих методов, чтобы ясно указать свои намерения.

Техника

- Если необходимое значение конструктору не предоставляется, следует прибегнуть к рефакторингу *Изменение объявления функции* (с. 170), чтобы добавить его в конструктор. Добавьте в конструктор вызов соответствующего метода установки значения поля.

  Если вы хотите удалить несколько методов установки значений, добавьте все их значения в конструктор за один раз. Это упростит последующие шаги.

- Удалите каждый вызов метода установки значения за пределами конструктора, используя вместо этого новый конструктор. Выполните тестирование после каждого удаления.

  Если вы не можете заменить вызов метода установки созданием нового объекта (поскольку обновляете совместно используемый ссылочный объект), откажитесь от рефакторинга.

- Примените рефакторинг *Встраивание функции* (с. 161) к методу установки. По возможности сделайте поле неизменяемым.

- Выполните тестирование.

Пример

У меня есть простой класс человека.

```
class Person...
get name()    {return this._name;}
set name(arg) {this._name = arg;}
get id()    {return this._id;}
set id(arg) {this._id = arg;}
```

В настоящее время новый объект создается с помощью кода, подобного показанному ниже.

```
const martin = new Person();
martin.name = "martin";
martin.id = "1234";
```

Имя человека может измениться после его создания, но его идентификатор меняться не должен. Чтобы ясно это указать, я хочу удалить метод установки идентификатора.

Мне все еще требуется начальная установка идентификатора, поэтому я использую рефакторинг *Изменение объявления функции* (с. 170), чтобы добавить ее в конструктор.

```
class Person...
constructor(id) {
  this.id = id;
}
```

Затем я изменяю сценарий создания таким образом, чтобы идентификатор устанавливался через конструктор.

```
const martin = new Person("1234");
martin.name = "martin";
~~martin.id = "1234";~~
```

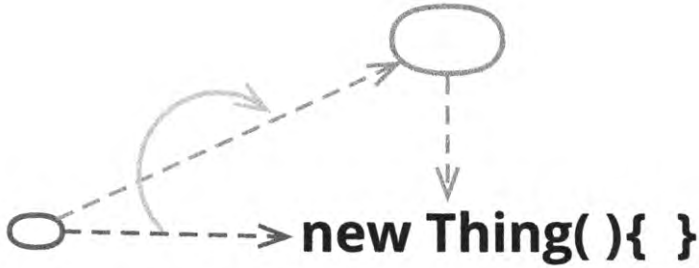
Я делаю это в каждом месте, где создается объект человека, и после каждого внесенного изменения выполняю тестирование.

Когда вся работа выполнена, к методу установки применяется рефакторинг *Встраивание функции* (с. 161).

```
class Person...
constructor(id) {
  this._id = id;
}
get name()    {return this._name;}
set name(arg) {this._name = arg;}
get id()    {return this._id;}
~~set id(arg) {this._id = arg;}~~
```

Замена конструктора фабричной функцией (Replace Constructor with Factory Function)

Бывший рефакторинг *Замена конструктора фабричным методом*




```
leadEngineer = new Employee(document.leadEngineer, 'E');
```

```
leadEngineer = createEngineer(document.leadEngineer);
```

Мотивация

Многие объектно-ориентированные языки содержат специальную функцию — конструктор, вызываемый для инициализации объекта. Клиенты обычно вызывают конструктор, когда хотят создать новый объект. Но зачастую конструкторы имеют определенные ограничения, которых нет для других, более общих функций. Так, конструктор Java должен возвращать экземпляр класса, для которого он был вызван, а значит, я не могу заменить его подклассом или прокси-классом в зависимости от среды или передаваемых параметров. Имя конструктора является фиксированным, так что я не могу использовать имя более понятное, чем назначаемое по умолчанию. Конструкторы часто требуется вызывать с помощью специального оператора (во многих языках — `new`), что затрудняет их применение в контекстах, ожидающих обычные функции.

Фабричная функция не имеет таких ограничений. Она обычно вызывает конструктор как часть своей реализации, но я могу заменить его чем-то иным и написать фабричную функцию так, как это требуется мне.

Техника

- Создайте фабричную функцию, тело которой представляет собой вызов конструктора.

- Замените каждый вызов конструктора вызовом фабричной функции.
- Выполните тестирование после внесения каждого изменения.
- Максимально ограничьте видимость конструктора.

Пример

В рассматриваемом быстром, но скучном примере используются разные виды служащих. Рассмотрим класс служащего:

```
class Employee...
constructor (name, typeCode) {
  this._name = name;
  this._typeCode = typeCode;
}

get name() {return this._name;}
get type() {
  return Employee.legalTypeCodes[this._typeCode];
}
static get legalTypeCodes() {
  return {"E": "Engineer", "M": "Manager", "S": "Salesman"};
}
```

Он используется в двух точках вызова:

```
Точка вызова...
candidate = new Employee(document.name, document.empType);
```

и

```
Точка вызова...
const leadEngineer = new Employee(document.leadEngineer, 'E');
```

Мой первый шаг состоит в создании фабричной функции. Ее тело представляет собой простое делегирование конструктору.

```
Верхний уровень...
function createEmployee(name, typeCode) {
  return new Employee(name, typeCode);
}
```

Затем я нахожу вызовы конструктора и по одному изменяю их так, чтобы вместо них использовалась фабричная функция.

Первый вызов очевиден:

```
Точка вызова...
candidate = createEmployee(document.name, document.empType);
```

Во втором случае я могу использовать новую фабричную функцию следующим образом:

```
Точка вызова...
const leadEngineer = createEmployee(document.leadEngineer, 'E');
```

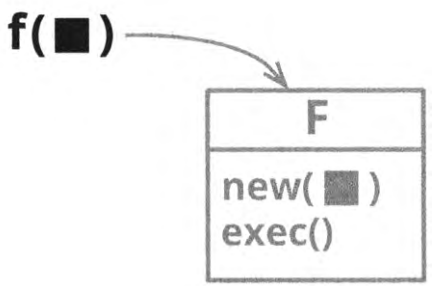
Но мне не нравится применение здесь кода типа — передача кода в виде литерала обычно плохо пахнет. Поэтому я предпочитаю создать новую фабричную функцию, имя которой будет указывать вид создаваемого ею служащего.

```
Точка вызова...
const leadEngineer = createEngineer(document.leadEngineer);

Верхний уровень...
function createEngineer(name) {
  return new Employee(name, 'E');
}
```

Замена функции командой (Replace Function with Command)

Бывший рефакторинг *Замена метода объектом методов*

Обратный к рефакторингу *Замена команды функцией* (с. 388)




```
function score(candidate, medicalExam, scoringGuide) {
  let result = 0;
  let healthLevel = 0;
  // Длинный код тела функции
}
```

```
class Scorer {
  constructor(candidate, medicalExam, scoringGuide) {
    this._candidate = candidate;
    this._medicalExam = medicalExam;
    this._scoringGuide = scoringGuide;
  }

  execute() {
    this._result = 0;
    this._healthLevel = 0;
    // Длинный код тела функции
  }
}
```

Мотивация

Функции — как автономные, так и связанные с объектами методы — являются одними из фундаментальных строительных блоков программирования. Но бывают случаи, когда полезно инкапсулировать функцию в свой собственный объект, который я называю "командным объектом" или просто **командой**. Такой объект в основном строится вокруг единственного метода, запрос и выполнение которого являются предназначением данного объекта.

Команда обеспечивает большую гибкость, чем простой механизм функции. Команды могут иметь дополнительные операции, такие как отмена действия и возврат к предыдущему состоянию. Я могу предоставить методы создания параметров команды для поддержания более богатого жизненного цикла; могу использовать различные настройки, применяя наследование и ловушки. Если я работаю на языке, в котором есть объекты, но нет функций первого класса[1], то большую часть их функциональных возможностей можно представить с использованием команд. Точно так же я могу использовать методы и поля, чтобы помочь разделить сложную функцию (даже на языке, в котором вложенные функции отсутствуют), и непосредственно вызывать эти методы во время тестирования и отладки.

Все описанное представляет собой веские причины для использования команд, и я должен быть готов, если понадобится, к рефакторингу функций в команды. Но не следует забывать, что, как обычно, эта гибкость имеет свою цену — а именно повышенную сложность. С учетом этого соображения при выборе между функцией первого класса и командой я в 95% случаев выберу функцию. Команду я использую только тогда, когда мне действительно нужны возможности, которые не могут обеспечить более простые подходы.

Как и многие другие термины в области программирования, термин *команда* оказывается чрезмерно перегруженным. В используемом в этом разделе контексте это объект, который инкапсулирует запрос, следуя проектному шаблону *Команда* [11]. Когда я говорю о команде в указанном смысле, я использую термин *объект команды* или *командный объект*, чтобы указать используемый контекст, а затем сокращаю термин просто до *команды*. Термин *команда* используется также в контексте принципа разделения команд и запросов [20], где команда представляет собой метод объекта, который изменяет наблюдаемое состояние. Я всегда стараюсь избегать использования термина *команда* в этом смысле, предпочитая ему термин *модификатор*.

---

[1] В информатике язык программирования имеет *функции первого класса*, если он рассматривает функции как объекты первого класса. В частности, это означает, что язык поддерживает передачу функций в качестве аргументов другим функциям, возврат их как результат других функций, присваивание их переменным или сохранение в структурах данных. — *Примеч. пер.*

Техника

- Создайте для функции пустой класс. Дайте ему имя, основанное на функции, с которой вы работаете.
- Примените рефакторинг *Перенос функции* (с. 244) для переноса функции в пустой класс.

  Сохраните исходную функцию как делегирующую (по крайней мере до конца рефакторинга).

  Соблюдайте все соглашения, имеющиеся в языке для именования команд. Если таковые соглашения отсутствуют, выберите для функции выполнения команды нейтральное общее имя, наподобие `execute` или `call`.
- Рассмотрите возможность создания поля для каждого аргумента и перенесите эти аргументы в конструктор.

Пример

Язык JavaScript имеет множество недостатков, но одним из замечательных решений при его разработке было создание функций как сущностей первого класса. Поэтому мне не нужно решать все проблемы создания команд для общих задач, с которыми пришлось бы иметь дело при работе на языках без этой возможности. Но бывают моменты, когда команда является единственно подходящим инструментом для выполнения работы.

Одним из таких случаев является разделение сложной функции на части, чтобы ее можно было легче понять и изменить. Чтобы показать реальную ценность этого рефакторинга, нужна длинная и сложная функция, но работа с ней заняла бы слишком много времени, не говоря уже о времени, которое потребовалось бы вам для чтения. Поэтому я вынужден рассмотреть в качестве примера функцию, которая достаточно коротка, чтобы не нуждаться в описываемом рефакторинге. Это функция подсчета баллов заявки на страховку.

```
function score(candidate, medicalExam, scoringGuide) {
  let result = 0;
  let healthLevel = 0;
  let highMedicalRiskFlag = false;

  if (medicalExam.isSmoker) {
    healthLevel += 10;
    highMedicalRiskFlag = true;
  }

  let certificationGrade = "regular";
  if (scoringGuide.stateWithLowCertification(
                            candidate.originState)) {
```

```
    certificationGrade = "low";
    result -= 5;
  }

  // И еще много похожего кода
  result -= Math.max(healthLevel - 5, 0);
  return result;
}
```

Я начинаю с создания пустого класса и применения рефакторинга *Перенос функции* (с. 244) для переноса функции в этот пустой класс.

```
function score(candidate, medicalExam, scoringGuide) {
  return new Scorer().execute(candidate, medicalExam,
                                         scoringGuide);
}

class Scorer {
  execute (candidate, medicalExam, scoringGuide) {
    let result = 0;
    let healthLevel = 0;
    let highMedicalRiskFlag = false;

    if (medicalExam.isSmoker) {
      healthLevel += 10;
      highMedicalRiskFlag = true;
    }

    let certificationGrade = "regular";
      if (scoringGuide.stateWithLowCertification(
                                candidate.originState)) {
      certificationGrade = "low";
      result -= 5;
    }

    // И еще много похожего кода
    result -= Math.max(healthLevel - 5, 0);
    return result;
  }
}
```

В большинстве случаев я предпочитаю передавать аргументы команде в конструкторе и иметь метод `execute`, который не принимает параметров. Хотя для простого сценария декомпозиции, такого как рассматриваемый, это менее важно, но все же это очень удобно, когда требуется манипулировать командой с более сложными жизненным циклом или настройками параметров. Различные классы команд могут иметь разные параметры, но быть смешаны вместе при постановке в очередь на выполнение.

Эти параметры можно обработать по одному.

```
function score(candidate, medicalExam, scoringGuide) {
  return new Scorer(candidate).execute(~~candidate~~, medicalExam,
                                       scoringGuide);
}

class Scorer...
constructor(candidate) {
  this._candidate = candidate;
}

execute(~~candidate~~, medicalExam, scoringGuide) {
  let result = 0;
  let healthLevel = 0;
  let highMedicalRiskFlag = false;

  if (medicalExam.isSmoker) {
    healthLevel += 10;
    highMedicalRiskFlag = true;
  }

  let certificationGrade = "regular";
  if (scoringGuide.stateWithLowCertification(
           this._candidate.originState)) {
    certificationGrade = "low";
    result -= 5;
  }

  // И еще много похожего кода
  result -= Math.max(healthLevel - 5, 0);
  return result;
}
```

Далее я продолжаю работать с другими параметрами.

```
function score(candidate, medicalExam, scoringGuide) {
  return new Scorer(candidate,medicalExam,scoringGuide).execute();
}

class Scorer...
constructor(candidate, medicalExam, scoringGuide){
  this._candidate = candidate;
  this._medicalExam = medicalExam;
  this._scoringGuide = scoringGuide;
}

execute () {
  let result = 0;
  let healthLevel = 0;
  let highMedicalRiskFlag = false;
```

```
  if (this._medicalExam.isSmoker) {
    healthLevel += 10;
    highMedicalRiskFlag = true;
  }

  let certificationGrade = "regular";
  if (this._scoringGuide.stateWithLowCertification(
           this._candidate.originState)) {
    certificationGrade = "low";
    result -= 5;
  }

  // И еще много похожего кода
  result -= Math.max(healthLevel - 5, 0);
  return result;
}
```

Этим завершается выполнение рефакторинга, но так как основная его цель — позволить мне разбивать сложные функции, намечу некоторые шаги для достижения этой цели. Следующий шаг состоит в том, чтобы заменить все локальные переменные полями. Эту замену я также выполняю по одной переменной за раз.

```
class Scorer...
constructor(candidate, medicalExam, scoringGuide){
  this._candidate = candidate;
  this._medicalExam = medicalExam;
  this._scoringGuide = scoringGuide;
}

execute () {
  this._result = 0;
  let healthLevel = 0;
  let highMedicalRiskFlag = false;

  if (this._medicalExam.isSmoker) {
    healthLevel += 10;
    highMedicalRiskFlag = true;
  }

  let certificationGrade = "regular";
  if (this._scoringGuide.stateWithLowCertification(
              this._candidate.originState)) {
    certificationGrade = "low";
    this._result -= 5;
  }

  // И еще много похожего кода
  this._result -= Math.max(healthLevel - 5, 0);
  return this._result;
}
```

Я повторяю эти действия для всех локальных переменных. (Это один из тех рефакторингов, который настолько прост, что я не дал ему собственную запись в каталоге рефакторингов, и теперь чувствую себя немного виноватым.)

```
class Scorer...
constructor(candidate, medicalExam, scoringGuide){
  this._candidate = candidate;
  this._medicalExam = medicalExam;
  this._scoringGuide = scoringGuide;
}

execute () {
  this._result = 0;
  this._healthLevel = 0;
  this._highMedicalRiskFlag = false;

  if (this._medicalExam.isSmoker) {
    this._healthLevel += 10;
    this._highMedicalRiskFlag = true;
  }

  this._certificationGrade = "regular";
  if (this._scoringGuide.stateWithLowCertification(
               this._candidate.originState)) {
    this._certificationGrade = "low";
    this._result -= 5;
}

  // И еще много похожего кода
  this._result -= Math.max(this._healthLevel - 5, 0);
  return this._result;
}
```

Теперь, когда я перенес все состояния функции в командный объект, можно использовать такой рефакторинг, как *Извлечение функции* (с. 152), не запутываясь во всех этих переменных и их областях видимости.

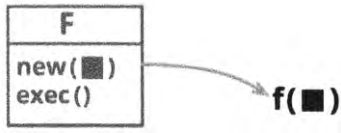
```
class Scorer...
execute () {
  this._result = 0;
  this._healthLevel = 0;
  this._highMedicalRiskFlag = false;

  this.scoreSmoking();
  this._certificationGrade = "regular";
  if (this._scoringGuide.stateWithLowCertification(
               this._candidate.originState)) {
    this._certificationGrade = "low";
    this._result -= 5;
  }
```

```
  // И еще много похожего кода
  this._result -= Math.max(this._healthLevel - 5, 0);
  return this._result;
}

scoreSmoking() {
  if (this._medicalExam.isSmoker) {
    this._healthLevel += 10;
    this._highMedicalRiskFlag = true;
  }
}
```

Это позволяет работать с командой так же, как с вложенной функцией. Действительно, в JavaScript применение вложенных функций было бы разумной альтернативой использованию команды. Тем не менее я все же применяю команды, отчасти потому, что я лучше с ними знаком, а отчасти потому, что с помощью команд легче писать тесты и отлаживать вызовы подфункций.

Замена команды функцией (Replace Command with Function)

Обратный к рефакторингу *Замена функции командой* (с. 381)




```
class ChargeCalculator {
  constructor (customer, usage){
    this._customer = customer;
    this._usage = usage;
  }
  execute() {
    return this._customer.rate * this._usage;
  }
}
```

```
function charge(customer, usage) {
  return customer.rate * usage;
}
```

Мотивация

Командные объекты предоставляют мощный механизм обработки сложных вычислений. Их можно легко разбить на отдельные методы, совместно использующие общее состояние через поля; для разных действий они могут быть вызваны разными способами; они могут поэтапно создавать свои данные. Но эта мощь имеет свою стоимость. А в большинстве случаев мне достаточно просто вызвать функцию и заставить ее делать свое дело. Если это так, и функция не слишком сложна, то цена командного объекта оказывается слишком велика, и его следует превратить в обычную функцию.

Техника

- Примените рефакторинг *Извлечение функции* (с. 152) к созданию команды и вызова метода выполнения команды.

  Это действие создает новую функцию, которая со временем заменит команду.

- Примените рефакторинг *Встраивание функции* (с. 161) к каждому методу, вызываемому методом выполнения команды.

  Если вспомогательная функция возвращает значение, сначала примените рефакторинг *Извлечение переменной* (с. 165) к ее вызову, а затем — рефакторинг *Встраивание функции* (с. 161).

- Воспользуйтесь рефакторингом *Изменение объявления функции* (с. 170) для того, чтобы поместить все параметры конструктора в метод выполнения команды.
- Для каждого поля измените обращение к нему в методе выполнения команды на использование параметра. Проведите тестирование после внесения каждого изменения.
- Встройте вызовы конструктора и метода выполнения команды в вызывающую функцию.
- Выполните тестирование.
- Примените рефакторинг *Удаление неработающего кода* (с. 283) к классу команды.

Пример

Я буду работать со следующим маленьким командным классом.

```
class ChargeCalculator {
  constructor (customer, usage, provider){
    this._customer = customer;
    this._usage = usage;
    this._provider = provider;
  }

  get baseCharge() {
    return this._customer.baseRate * this._usage;
  }

  get charge() {
    return this.baseCharge + this._provider.connectionCharge;
  }
}
```

Он используется кодом наподобие следующего.

```
Точка вызова...
monthCharge =
    new ChargeCalculator(customer, usage, provider).charge;
```

Командный класс невелик и достаточно прост, чтобы его было лучше превратить в функцию.

Начнем с применения рефакторинга *Извлечение функции* (с. 152) для обертки создания и вызова класса.

```
Точка вызова...
monthCharge = charge(customer, usage, provider);

Верхний уровень...
function charge(customer, usage, provider) {
  return new ChargeCalculator(customer,usage,provider).charge;
}
```

Следует решить, как работать с любыми вспомогательными функциями, в данном случае — с функцией `baseCharge`. Мой обычный подход к функции, возвращающей значение, заключается в том, чтобы сначала применить рефакторинг *Извлечение переменной* (с. 165) к этому значению.

```
class ChargeCalculator...
get baseCharge() {
  return this._customer.baseRate * this._usage;
}

get charge() {
  const baseCharge = this.baseCharge;
  return baseCharge + this._provider.connectionCharge;
}
```

Затем я применяю рефакторинг *Встраивание функции* (с. 161) к вспомогательной функции.

```
class ChargeCalculator...
get charge() {
  const baseCharge = this._customer.baseRate * this._usage;
  return baseCharge + this._provider.connectionCharge;
}
```

Теперь все вычисления сосредоточены в одной функции, так что мой следующий шаг — перенести переданные в конструктор данные в метод `main`. Сначала я использую рефакторинг *Изменение объявления функции* (с. 170) для того, чтобы добавить все параметры конструктора в метод `charge`.

```
class ChargeCalculator...
constructor (customer, usage, provider){
  this._customer = customer;
  this._usage = usage;
  this._provider = provider;
}

charge(customer, usage, provider) {
  const baseCharge = this._customer.baseRate * this._usage;
  return baseCharge + this._provider.connectionCharge;
}

Верхний уровень...
function charge(customer, usage, provider) {
  return new ChargeCalculator(customer, usage, provider)
                    .charge(customer, usage, provider);
}
```

Теперь можно изменить тело `charge` таким образом, чтобы в нем использовались переданные параметры. Я могу обрабатывать параметры по одному.

```
class ChargeCalculator...
constructor (customer, usage, provider){
  this._customer = customer;
  this._usage = usage;
  this._provider = provider;
}

charge(customer, usage, provider) {
  const baseCharge = customer.baseRate * this._usage;
  return baseCharge + this._provider.connectionCharge;
}
```

Мне не нужно удалять присваивание полю `this._customer` в конструкторе, так как оно будет просто проигнорировано. Но я предпочитаю сделать это, чтобы получить сбой теста, если изменение поля в параметре окажется пропущенным.

(И если тест проходит успешно, я должен рассмотреть возможность добавления нового теста.)

Я повторяю эти действия для других параметров, получая в конечном итоге

```
class ChargeCalculator...
charge(customer, usage, provider) {
  const baseCharge = customer.baseRate * usage;
  return baseCharge + provider.connectionCharge;
}
```

Последним можно выполнить встраивание в функцию верхнего уровня `charge`. Это разновидность рефакторинга *Встраивание функции* (с. 161), так как он выполняет встраивание как конструктора, так и вызова метода.

```
Верхний уровень...
function charge(customer, usage, provider) {
  const baseCharge = customer.baseRate * usage;
  return baseCharge + provider.connectionCharge;
}
```

Теперь командный класс представляет собой мертвый код, так что я устраиваю ему торжественные похороны с помощью рефакторинга *Удаление неработающего кода* (с. 283).

Глава 12

Работа с наследованием

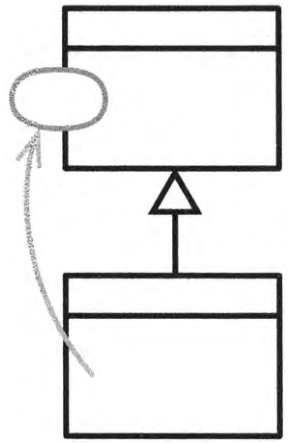
В последней главе я расскажу об одной из самых известных возможностей объектно-ориентированного программирования — наследовании. Как и любой мощный механизм, он одновременно и очень полезен, и его легко использовать неправильно. Зачастую его наличие просто трудно заметить, пока не столкнешься с неверным его использованием.

Функциональные возможности зачастую должны перемещаться вверх или вниз по иерархии наследования. С таким перемещением связано несколько рефакторингов: *Подъем метода* (с. 394), *Подъем поля* (с. 397), *Подъем тела конструктора* (с. 399), *Опускание метода* (с. 403) и *Опускание поля* (с. 404). Можно добавлять и удалять классы из иерархии с помощью рефакторингов *Извлечение суперкласса* (с. 418), *Удаление подкласса* (с. 413) и *Свертывание иерархии* (с. 423). При необходимости можно добавить подкласс, чтобы заменить поле, используемое для запуска другого поведения, основанного на его значении. Сделать это позволяет рефакторинг *Замена кода типа подклассами* (с. 405).

Наследование — мощный инструмент, но иногда оно используется в неподходящем месте (или место, где оно используется, становится неправильным). В этом случае можно воспользоваться рефакторингами *Замена подкласса делегатом* (с. 424) или *Замена суперкласса делегатом* (с. 443), чтобы превратить наследование в делегирование.

Подъем метода (Pull Up Method)

Обратный к рефакторингу *Опускание метода* (с. 404)



```
class Employee {...}

class Salesman extends Employee {
  get name() {...}
}

class Engineer extends Employee {
  get name() {...}
}
```

```
class Employee {
  get name() {...}
}

class Salesman extends Employee {...}
class Engineer extends Employee {...}
```

Мотивация

Устранение дублирования поведения — важная работа. Хотя два продублированных метода могут прекрасно работать в неизменном виде, они представляют собой питательную среду для возникновения ошибок в будущем. При наличии дублирования всегда есть риск, что при внесении изменений в один метод второй будет забыт и пропущен. Находить дубликаты обычно достаточно трудно.

Простейший пример применения данного рефакторинга — когда тела двух методов одинаковы настолько, что это указывает на выполненные копирование и вставку. Конечно, далеко не всегда ситуация столь тривиальна. Можно просто выполнить рефакторинг и посмотреть, успешно ли пройдут тесты, но тогда придется полностью положиться на них. Я обычно считаю, что стоит поискать различия в методах; часто они указывают на поведение, которое я забыл протестировать.

Зачастую рассматриваемый рефакторинг применяется после внесения других изменений. Можно обнаружить два метода в разных классах, которые можно параметризовать так, что это, по сути, приведет к одному и тому же методу. В этом случае наименьшим шагом является отдельное применение рефакторинга *Параметризация функции* (с. 355) с последующим рефакторингом *Подъем метода* (с. 394).

Самое неприятное в данном рефакторинге — то, что тело метода может обращаться к сущностям, находящимся в подклассе, а не в суперклассе. Если такое случилось, требуется сначала применить рефакторинги *Подъем поля* (с. 397) и *Подъем метода* к этим элементам.

Если есть два похожих, но разных метода, можно попробовать прибегнуть к рефакторингу *Формирование шаблонного метода* [25].

Техника

- Изучите методы и убедитесь, что они идентичны.

  Если методы решают одну и ту же задачу, но не идентичны, примените к ним рефакторинги, делающие тела методов полностью идентичными.
- Убедитесь, что все вызовы методов и ссылки на поля внутри тела метода относятся к функциям, которые могут быть вызваны из суперкласса.
- Если методы имеют разные сигнатуры, примените рефакторинг *Изменение объявления функции* (с. 170), чтобы получить сигнатуру, которую вы хотите использовать в суперклассе.
- Создайте в суперклассе новый метод и скопируйте в него тело одного из методов.
- Выполните статические проверки.
- Удалите один метод подкласса.
- Выполните тестирование.
- Продолжайте удаление методов подклассов и тестирование, пока не удалите их все.

Пример

У меня есть два метода подклассов, выполняющих одну и ту же работу.

```
class Employee extends Party...
  get annualCost() {
return this.monthlyCost * 12;
}

class Department extends Party...
get totalAnnualCost() {
  return this.monthlyCost * 12;
}
```

Я смотрю на оба класса и вижу, что они обращаются к свойству `monthlyCost`, которое не определено в суперклассе, но присутствует в обоих подклассах. Поскольку я работаю с динамическим языком — все в порядке; но если бы я работал со статическим языком, мне нужно было бы определить абстрактный метод для `Party`.

Рассматриваемые методы имеют разные имена, поэтому я прибегаю к рефакторингу *Изменение объявления функции* (с. 170), чтобы сделать их одинаковыми.

```
class Department...
get annualCost() {
  return this.monthlyCost * 12;
}
```

Я копирую метод одного из подклассов и вставляю его в суперкласс.

```
class Party...
get annualCost() {
  return this.monthlyCost * 12;
}
```

В случае статического языка программирования я бы выполнил компиляцию, чтобы убедиться, что все ссылки в порядке. Здесь это мне не поможет, поэтому я сначала удаляю `annualCost` из `Employee`, выполняю тестирование, а затем удаляю это поле из `Department`.

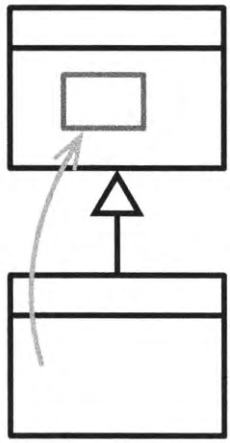
На этом рефакторинг завершен, но остается вопрос: `annualCost` вызывает функцию `monthlyCost`, но `monthlyCost` в классе `Party` не является видимым. Все это работает, потому что JavaScript — язык динамический, но имеет смысл указать, что подклассы `Party` должны предоставлять реализацию `monthlyCost`, особенно если позднее будут добавляться новые подклассы. Хороший способ обеспечить такой сигнал — метод-ловушка, подобный показанному ниже.

```
class Party...
get monthlyCost() {
  throw new SubclassResponsibilityError();
}
```

Я называю такую ошибку *ошибкой ответственности подкласса*, поскольку именно это имя использовалось в Smalltalk.

Подъем поля (Pull Up Field)

Обратный к рефакторингу *Опускание поля* (с. 404)



```
class Employee {...} // Java

class Salesman extends Employee {
  private String name;
}

class Engineer extends Employee {
  private String name;
}
```

```
class Employee {
  protected String name;
}

class Salesman extends Employee {...}
class Engineer extends Employee {...}
```

Мотивация

Если подклассы разрабатываются независимо или объединяются посредством рефакторинга, часто обнаруживается, что они дублируют функциональные возможности. В частности, дубликатами могут быть некоторые поля. Такие поля иногда (но не всегда) имеют похожие имена. Единственный способ узнать, так ли это на самом деле — изучить, как используются данные поля. Если они используются одинаково — их можно перенести в суперкласс.

Делая это, я уменьшаю дублирование двумя способами — я удаляю дубликат объявления данных, а затем могу переместить поведение, использующее это поле, из подклассов в суперкласс.

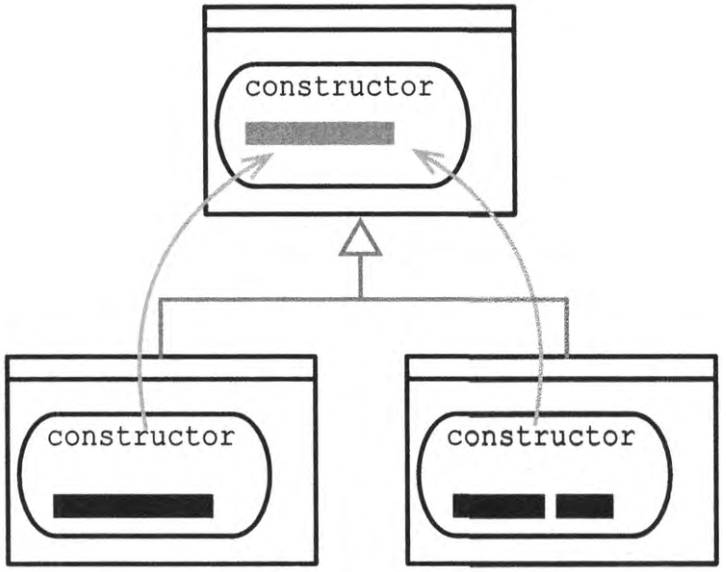
Многие динамические языки не определяют поля как часть определения класса — вместо этого поля появляются при первом присваивании им значений. В этом случае подъем поля по сути является следствием рефакторинга *Подъем тела конструктора* (с. 399).

Техника

- Изучите всех пользователей поля-кандидата, чтобы убедиться, что поля используются одинаково.
- Если поля имеют разные имена, используйте рефакторинг *Переименование поля* (с. 289), чтобы присвоить им одинаковые имена.
- Создайте новое поле в суперклассе.

  Новое поле должно быть доступно подклассам (объявлено как `protected` в распространенных языках программирования).
- Удалите поля в подклассах.
- Выполните тестирование.

Подъем тела конструктора (Pull Up Constructor Body)




```
class Party {...}

class Employee extends Party {
  constructor(name, id, monthlyCost) {
    super();
    this._id = id;
    this._name = name;
    this._monthlyCost = monthlyCost;
  }
}
```

```
class Party {
  constructor(name){
    this._name = name;
  }
}

class Employee extends Party {
  constructor(name, id, monthlyCost) {
    super(name);
    this._id = id;
    this._monthlyCost = monthlyCost;
  }
}
```

Мотивация

Конструкторы — штука хитрая и сложная. Это не совсем нормальные методы, поэтому я ограничен в том, что могу делать с ними.

Обнаружив методы подкласса с общим поведением, я вначале использую рефакторинг *Извлечение функции* (с. 152), а затем — *Подъем метода* (с. 394), который красиво перемещает метод в суперкласс. Но в случае конструкторов все не так просто, потому что для них есть особые правила, что можно делать и в каком порядке, так что мне потребуется немного другой подход.

Если этот рефакторинг начинает становиться запутанным, я предпочитаю перейти к рефакторингу *Замена конструктора фабричной функцией* (с. 379).

Техника

- Определите конструктор суперкласса, если таковой еще не существует. Убедитесь, что он вызывается конструкторами подкласса.
- Используйте рефакторинг *Перемещение инструкций* (с. 269), чтобы перенести любые общие инструкции, разместив их сразу после вызова `super()`.
- Удалите общий код из всех подклассов и поместите его в суперкласс. Добавьте к вызову `super()` все параметры конструктора, к которым есть обращение в общем коде.
- Выполните тестирование.
- Если имеется какой-либо общий код, который не может быть перенесен в начало конструктора, используйте рефакторинг *Извлечение функции* (с. 152), за которым следует рефакторинг *Подъем метода* (с. 394).

Пример

Начну со следующего кода.

```
class Party {}

class Employee extends Party {
  constructor(name, id, monthlyCost) {
    super();
    this._id = id;
    this._name = name;
    this._monthlyCost = monthlyCost;
  }
  // Остальная часть класса...

class Department extends Party {
  constructor(name, staff){
    super();
```

```
    this._name = name;
    this._staff = staff;
  }
  // Остальная часть класса...
```

Здесь общий код — присваивание имени. Я использую рефакторинг *Перемещение инструкций* (с. 269), чтобы переместить присваивание в `Employee` рядом с вызовом `super()`.

```
class Employee extends Party {
  constructor(name, id, monthlyCost) {
    super();
    this._name = name;
    this._id = id;
    this._monthlyCost = monthlyCost;
  }
  // Остальная часть класса...
```

После тестирования я перемещаю общий код в суперкласс. Поскольку этот код содержит ссылку на аргумент конструктора, я передаю его в качестве параметра.

```
class Party...
constructor(name){
  this._name = name;
}

class Employee...
constructor(name, id, monthlyCost) {
  super(name);
  this._id = id;
  this._monthlyCost = monthlyCost;
}

class Department...
constructor(name, staff){
  super(name);
  this._staff = staff;
}
```

После выполнения тестирования работа завершена.

В большинстве случаев конструктор будет работать следующим образом: сначала выполнит общие элементы поведения (с помощью вызова `super()`), а затем — дополнительную работу, необходимую подклассу. Иногда, однако, некоторое общее поведение может появиться позже.

Рассмотрим следующий пример:

```
class Employee...
constructor (name) {...}

get isPrivileged() {...}
```

```
assignCar() {...}

class Manager extends Employee...
constructor(name, grade) {
  super(name);
  this._grade = grade;

  // Это делает каждый подкласс:
  if (this.isPrivileged) this.assignCar();
}

get isPrivileged() {
  return this._grade > 4;
}
```

Проблема заключается в том, что вызов `isPrivileged` не может быть выполнен до тех пор, пока не будет присвоено значение полю `grade`, а это может быть сделано только в подклассе.

В этом случае для общего кода я применяю рефакторинг *Извлечение функции* (с. 152).

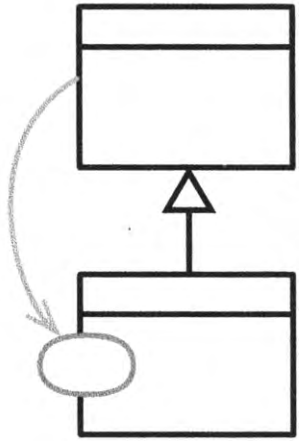
```
class Manager...
constructor(name, grade) {
  super(name);
  this._grade = grade;
  this.finishConstruction();
}

finishConstruction() {
  if (this.isPrivileged) this.assignCar();
}
```

Затем использую рефакторинг *Подъем метода* (с. 394) для переноса метода в суперкласс.

```
class Employee...
finishConstruction() {
  if (this.isPrivileged) this.assignCar();
}
```

Опускание метода (Push Down Method)

Обратный к рефакторингу *Подъем метода* (с. 394)



```
class Employee {
  get quota {...}
}

class Engineer extends Employee {...}
class Salesman extends Employee {...}
```

```
class Employee {...}
class Engineer extends Employee {...}
class Salesman extends Employee {
  get quota {...}
}
```

Мотивация

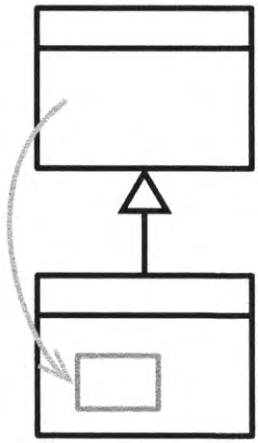
Если метод имеет отношение только к одному подклассу (или небольшому количеству подклассов), удаление его из суперкласса и размещение только в подклассе (подклассах) делает это отношение более понятным. Этот рефакторинг возможен только в том случае, когда вызывающий код знает, что он работает с определенным подклассом; в противном случае следует использовать рефакторинг *Замена условной инструкции полиморфизмом* (с. 317) с некоторым плацебо-поведением в суперклассе.

Техника

- Скопируйте метод в каждый подкласс, который в нем нуждается.
- Удалите метод из суперкласса.
- Выполните тестирование.
- Удалите метод из каждого суперкласса, в котором он не нужен.
- Выполните тестирование.

Опускание поля (Push Down Field)

Обратный к рефакторингу *Подъем поля* (с. 397)



```
class Employee { // Java
  private String quota;
}

class Engineer extends Employee {...}
class Salesman extends Employee {...}
```

```
class Employee {...}
class Engineer extends Employee {...}

class Salesman extends Employee {
  protected String quota;
}
```

Мотивация

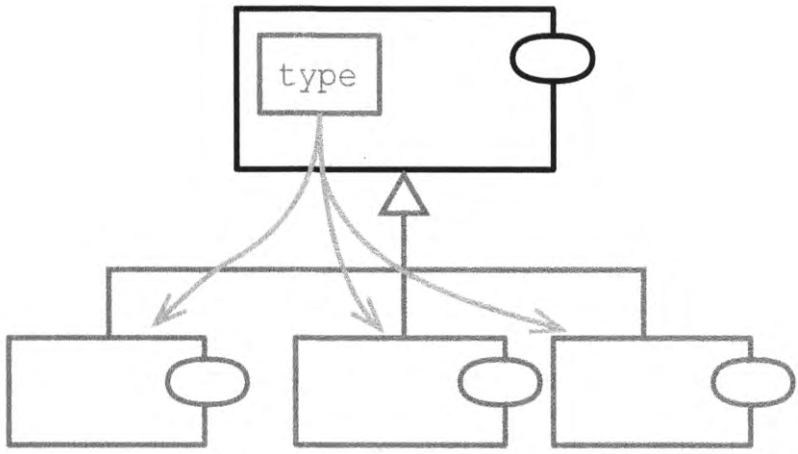
Если поле используется только одним подклассом (или небольшой долей подклассов), я перемещаю его в эти подклассы.

Техника

- Объявите поле в подклассах, нуждающихся в нем.
- Удалите поле из суперкласса.
- Выполните тестирование.
- Удалите поле из каждого суперкласса, в котором оно не нужно.
- Выполните тестирование.

Замена кода типа подклассами (Replace Type Code with Subclasses)

Поглощает *Замена кода типа состоянием/стратегией*
Поглощает *Извлечение подкласса*
Обратный к рефакторингу *Удаление подкласса* (с. 413)




```
function createEmployee(name, type) {
  return new Employee(name, type);
}
```

```
function createEmployee(name, type) {
  switch (type) {
    case "engineer": return new Engineer(name);
    case "salesman": return new Salesman(name);
    case "manager": return new Manager (name);
  }
```

Мотивация

Программные системы часто должны представлять разные виды схожих вещей. Я могу классифицировать сотрудников по выполняемой работе (инженер, менеджер, продавец), а заказы по приоритету (срочный, обычный). Простейший инструмент для обработки такой разновидности — поле кода типа, которое, в зависимости от используемого языка, может быть перечислением, символом, строкой или числом. Часто этот код типа поступает из внешней службы, которая предоставляет мне данные, с которыми я работаю.

В большинстве случаев все, что мне нужно — такой код типа. Но есть пара ситуаций, когда я мог бы сделать что-то большее, и это большее — подклассы. В подклассах особенно привлекательны две вещи. Во-первых, они позволяют использовать полиморфизм для обработки условной логики. Я считаю это наиболее полезным, когда у меня есть несколько функций, которые вызывают различное поведение в зависимости от значения кода типа. Используя подклассы, я могу применить к этим функциям рефакторинг *Замена условной инструкции полиморфизмом* (с. 317).

Во втором случае у меня есть поля или методы, которые действительны только для определенных значений кода типа, например квота продаж, применимая только к коду типа "продавец". В такой ситуации я могу создать подкласс и применить рефакторинг *Опускание поля* (с. 404). Хотя я могу включить логику проверки, гарантирующую, что поле используется только для правильного значения кода типа, использование подкласса делает эту взаимосвязь более явной.

При использовании рефакторинга *Замена кода типа подклассами* нужно задуматься, применять его непосредственно к рассматриваемому классу или к самому коду типа. Делать ли инженера подтипом сотрудника, или я должен дать сотруднику свойство типа сотрудника, которое может иметь подтипы для инженера и менеджера? Применение прямого подкласса проще, но я не смогу использовать его для типа работы, если таковой потребуется мне в дальнейшем. Я не могу задействовать непосредственные подклассы, если это изменяемый тип. Для того чтобы перенести подклассы в свойство типа сотрудника, можно применить рефакторинг *Замена примитива объектом* (с. 221) к коду типа, чтобы создать класс типа сотрудника, а затем выполнить рефакторинг *Замена кода типа подклассами* для этого нового класса.

Техника

- Самоинкапсулируйте поле кода типа.
- Выберите одно значение кода типа. Создайте подкласс для этого кода типа. Перекройте метод получения кода типа так, чтобы он возвращал литеральное значение кода типа.
- Создайте логику выбора, отображающую параметр кода типа на новый подкласс.

  При прямом наследовании используйте рефакторинг *Замена конструктора фабричной функцией* (с. 379) и поместите логику выбора в фабрику. При косвенном наследовании логика выбора может оставаться в конструкторе.
- Выполните тестирование.
- Повторите создание подкласса и добавление в логику выбора для каждого значения кода типа. Выполните тестирование после внесения каждого изменения.
- Удалите поле кода типа.
- Выполните тестирование.
- Примените рефакторинги *Опускание метода* (с. 403) и *Замена условной инструкции полиморфизмом* (с. 317) ко всем методам, которые используют методы доступа к коду типа. После выполнения всех замен методы доступа к кодам типа можно удалить.

Пример

Я начну со следующего примера чрезмерно занятого сотрудника.

```
class Employee...
constructor(name, type){
  this.validateType(type);
  this._name = name;
  this._type = type;
}
validateType(arg) {
  if (!["engineer", "manager", "salesman"].includes(arg))
    throw new Error(`Employee cannot be of type ${arg}`);
}

toString() {return `${this._name} (${this._type})`;}
```

Мой первый шаг состоит в применении рефакторинга *Инкапсуляция переменной* (с. 178) для самоинкапсуляции кода типа.

```
class Employee...
get type() {return this._type;}
toString() {return `${this._name} (${this.type})`;}
```

Обратите внимание: `toString` использует новый метод доступа путем удаления подчеркивания.

Я выбираю один код типа, инженер, с которого начну работу. Я использую прямое наследование, создавая подкласс класса работника. Подкласс сотрудника прост — все, что он делает, — это перекрывает метод доступа к коду типа соответствующим литеральным значением.

```
class Engineer extends Employee {
  get type() {return "engineer";}
}
```

Хотя конструкторы JavaScript могут возвращать другие объекты, если я попытаюсь поместить в конструктор логику выбора, получится неразбериха, поскольку эта логика переплетается с инициализацией поля. Поэтому я использую рефакторинг *Замена конструктора фабричной функцией* (с. 379), чтобы создать для нее новое пространство.

```
function createEmployee(name, type) {
  return new Employee(name, type);
}
```

Чтобы использовать новый подкласс, я добавляю логику выбора в фабрику.

```
function createEmployee(name, type) {
  switch (type) {
    case "engineer": return new Engineer(name, type);
  }
  return new Employee(name, type);
}
```

Я выполняю тестирование, чтобы убедиться, что все сработало правильно. Но, поскольку я параноик, затем я изменяю возвращаемое значение перекрытия в классе инженера и снова выполняю тестирование, чтобы убедиться, что тест не проходит. Таким образом, можно убедиться, что этот подкласс используется. Я исправляю возвращаемое значение и продолжаю работу с другими случаями. Их можно обрабатывать по одному, выполняя тестирование после каждого изменения.

```
class Salesman extends Employee {
  get type() {return "salesman";}
}

class Manager extends Employee {
  get type() {return "manager";}
}
```

```
function createEmployee(name, type) {
  switch (type) {
    case "engineer": return new Engineer(name, type);
    case "salesman": return new Salesman(name, type);
    case "manager" : return new Manager (name, type);
  }
  return new Employee(name, type);
}
```

Закончив со всеми типами сотрудников, можно удалить поле кода типа и метод получения суперкласса (поля и методы в подклассах остаются).

```
class Employee...
constructor(name, type){
  this.validateType(type);
  this._name = name;
  this._type = type;
}

get type() {return this._type;}
toString() {return `${this._name} (${this.type})`;}
```

После тестирования, которое позволяет убедиться, что все по-прежнему хорошо работает, можно удалить логику проверки, поскольку конструкция `switch` фактически выполняет ту же самую работу.

```
class Employee...
constructor(name, type){
  this.validateType(type);
  this._name = name;
}

function createEmployee(name, type) {
  switch (type) {
  case "engineer": return new Engineer(name, type);
  case "salesman": return new Salesman(name, type);
  case "manager": return new Manager (name, type);
  default: throw new Error(`Employee cannot be of type ${type}`);
  }
  return new Employee(name, type);
}
```

Аргумент типа конструктора теперь бесполезен, поэтому он становится жертвой рефакторинга *Изменение объявления функции* (с. 170).

```
class Employee...
constructor(name, type){
  this._name = name;
}
```

```
function createEmployee(name, type) {
  switch (type) {
    case "engineer": return new Engineer(name~~, type~~);
    case "salesman": return new Salesman(name~~, type~~);
    case "manager" : return new Manager (name~~, type~~);
    default: throw new Error(`Employee cannot be of type ${type}`);
  }
}
```

У меня все еще есть средства доступа к коду типа в подклассах — `get type`. Обычно их также хочется удалить, но это может занять некоторое время из-за наличия других методов, которые зависят от них. Чтобы справиться с ними, я буду использовать рефакторинги *Замена условной инструкции полиморфизмом* (с. 317) и *Опускание метода* (с. 403). В какой-то момент у меня больше не будет кода, использующего методы доступа к `type`, поэтому я воспользуюсь мизерикордом *Удаление неработающего кода* (с. 283).

Пример: использование косвенного наследования

Давайте вернемся к начальному случаю, но в этот раз у меня уже есть подклассы для сотрудников, занятых полный и неполный рабочий день, поэтому я не могу образовать подкласс от `Employee` для кодов типов. Еще одна причина не использовать прямое наследование — сохранение возможности изменять тип работника.

```
class Employee...
constructor(name, type){
  this.validateType(type);
  this._name = name;
  this._type = type;
}

validateType(arg) {
  if (!["engineer", "manager", "salesman"].includes(arg))
    throw new Error(`Employee cannot be of type ${arg}`);
}

get type() {return this._type;}
set type(arg) {this._type = arg;}

get capitalizedType() {
  return this._type.charAt(0).toUpperCase() +
    this._type.substr(1).toLowerCase();
}

toString() {
  return `${this._name} (${this.capitalizedType})`;
}
```

Мой первый шаг состоит в том, чтобы применить рефакторинг *Замена примитива объектом* (с. 221) к коду типа.

```
class EmployeeType {
  constructor(aString) {
    this._value = aString;
  }
  toString() {return this._value;}
}

class Employee...
constructor(name, type){
  this.validateType(type);
  this._name = name;
  this.type = type;
}

validateType(arg) {
  if (!["engineer", "manager", "salesman"].includes(arg))
    throw new Error(`Employee cannot be of type ${arg}`);
}

get typeString() {return this._type.toString();}
get type()       {return this._type;}
set type(arg)    {this._type = new EmployeeType(arg);}

get capitalizedType() {
  return this.typeString.charAt(0).toUpperCase()
    + this.typeString.substr(1).toLowerCase();
}

toString() {
  return `${this._name} (${this.capitalizedType})`;
}
```

Затем я применяю обычную технику рефакторинга *Замена кода типа подклассами* к типам сотрудников.

```
class Employee...
set type(arg) {this._type = Employee.createEmployeeType(arg);}

static createEmployeeType(aString) {
  switch(aString) {
    case "engineer": return new Engineer();
    case "manager": return new Manager ();
    case "salesman": return new Salesman();
    default: throw
        new Error(`Employee cannot be of type ${aString}`);
  }
}
```

```
class EmployeeType {
}

class Engineer extends EmployeeType {
  toString() {return "engineer";}
}

class Manager extends EmployeeType {
  toString() {return "manager";}
}

class Salesman extends EmployeeType {
  toString() {return "salesman";}
}
```

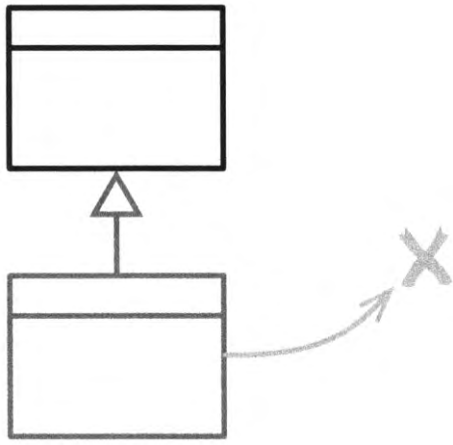
Если на этом остановиться, можно было бы удалить пустой `EmployeeType`. Но я предпочитаю оставить его на месте, так как он делает явной связь между различными подклассами. Это также удобное место для переноса сюда другого поведения, как, например, логика использования заглавных букв (пример которой в иллюстративных целях приведен ниже).

```
class Employee...
toString() {
  return `${this._name} (${this.type.capitalizedName})`;
}

class EmployeeType...
get capitalizedName() {
  return this.toString().charAt(0).toUpperCase()
    + this.toString().substr(1).toLowerCase();
}
```

Для тех, кто знаком с первым изданием книги, этот пример по сути является примером рефакторинга *Замена кода типа состоянием/стратегией*. Сейчас я рассматриваю этот рефакторинг как рефакторинг *Замена кода типа подклассами* с использованием косвенного наследования, а потому не считаю нужным посвящать ему отдельную запись в каталоге рефакторингов (тем более что мне никогда не нравилось это название).

Удаление подкласса (Remove Subclass)

Бывший рефакторинг *Замена подкласса полями*
Обратный к рефакторингу *Замена кода типа подклассами* (с. 405)



```
class Person {
  get genderCode() {return "X";}
}
class Male extends Person {
  get genderCode() {return "M";}
}
class Female extends Person {
  get genderCode() {return "F";}
}
```

```
class Person {
  get genderCode() {return this._genderCode;}
}
```

Мотивация

Подклассы полезны. Они поддерживают изменения в структуре данных и полиморфное поведение. Подклассы являются хорошим способом программирования на основе различий. Но по мере развития программной системы подклассы могут терять свою ценность, поскольку поддерживаемые ими вариации перемещаются в другие места или удаляются вовсе. Иногда подклассы добавляются в

ожидании функциональных возможностей, которые никогда не будут созданы, или будут созданы так, что им не потребуются подклассы.

Подкласс, который делает слишком мало, влечет за собой затраты на понимание, которое больше не требуется. Когда наступает такое время, лучше удалить подкласс, заменив его полем в суперклассе.

Техника

- Примените рефакторинг *Замена конструктора фабричной функцией* (с. 379) к конструктору подкласса.

  Если клиенты конструкторов используют поле данных, чтобы решить, какой именно подкласс создать, поместите эту логику принятия решения в метод фабрики суперкласса.
- Если некоторый код проверяет типы подклассов, примените рефакторинг *Извлечение функции* (с. 152) к проверке типа, а затем рефакторинг *Перенос функции* (с. 244) для его перемещения в суперкласс. Выполните тестирование после каждого изменения.
- Создайте поле для представления типа подкласса.
- Измените методы, которые обращаются к подклассу, чтобы они использовали новое поле типа.
- Удалите подкласс.
- Выполните тестирование.

Часто этот рефакторинг используется сразу для группы подклассов — в этом случае сначала выполните шаги по их инкапсуляции (добавьте фабричную функцию, переместите проверки типа), а затем по одному добавьте их в суперкласс.

Пример

Я начну с такого набора подклассов.

```
class Person...
constructor(name) {
  this._name = name;
}
get name() {return this._name;}
get genderCode() {return "X";}
// Прочий код

class Male extends Person {
  get genderCode() {return "M";}
}
```

```
class Female extends Person {
  get genderCode() {return "F";}
}
```

Если это все, что делает подкласс, то он не стоит этих строчек. Но прежде чем удалить показанные подклассы, обычно стоит проверить, нет ли в клиентском коде какого-либо поведения, зависимого от подкласса, которое могло бы нас остановить. В данном случае я не нахожу ничего, что требовало бы сохранения подклассов.

```
Код клиента...
const numberOfMales = people.filter(p => p instanceof Male).length;
```

Всякий раз, прежде чем изменить представление чего-то, я сначала пытаюсь инкапсулировать текущее представление, чтобы минимизировать воздействие на любой клиентский код. Когда дело доходит до создания подклассов, то инкапсуляция заключается в использовании рефакторинга *Замена конструктора фабричной функцией* (с. 379). В данном случае существует несколько способов создать фабрику.

Наиболее прямой и непосредственный вариант — создать фабричный метод для каждого конструктора.

```
function createPerson(name) {
  return new Person(name);
}
function createMale(name) {
  return new Male(name);
}
function createFemale(name) {
  return new Female(name);
}
```

Однако подобные объекты часто загружаются из источника, который использует код пола непосредственно.

```
function loadFromInput(data) {
  const result = [];
  data.forEach(aRecord => {
    let p;
    switch (aRecord.gender) {
      case 'M': p = new Male(aRecord.name); break;
      case 'F': p = new Female(aRecord.name); break;
      default: p = new Person(aRecord.name);
    }
    result.push(p);
  });
  return result;
}
```

В таких случаях мне кажется, что лучше всего применить к логике выбора создаваемого класса рефакторинг *Извлечение функции* (с. 152) и сделать ее фабричной функцией.

```
function createPerson(aRecord) {
  let p;
  switch (aRecord.gender) {
    case 'M': p = new Male(aRecord.name); break;
    case 'F': p = new Female(aRecord.name); break;
    default: p = new Person(aRecord.name);
  }
  return p;
}

function loadFromInput(data) {
  const result = [];
  data.forEach(aRecord => {
    result.push(createPerson(aRecord));
  });
  return result;
}
```

Раз уж я здесь, разберусь с этими двумя функциями. В функции `createPerson` воспользуюсь рефакторингом *Встраивание переменной* (с. 169):

```
function createPerson(aRecord) {
  switch (aRecord.gender) {
    case 'M': return new Male   (aRecord.name);
    case 'F': return new Female(aRecord.name);
    default:  return new Person(aRecord.name);
  }
}
```

а для функции `loadFromInput` у меня припасен рефакторинг *Замена цикла конвейером* (с. 278):

```
function loadFromInput(data) {
  return data.map(aRecord => createPerson(aRecord));
}
```

Фабрика инкапсулирует создание подклассов, но в коде имеется еще и использование `instanceof`, что всегда несколько попахивает. Я применяю к проверке типа рефакторинг *Извлечение функции* (с. 152).

```
Код клиента...
const numberOfMales = people.filter(p => isMale(p)).length;
function isMale(aPerson) {return aPerson instanceof Male;}
```

Затем использую рефакторинг *Перенос функции* (с. 244) для переноса ее в `Person`.

```
class Person...
get isMale() {return this instanceof Male;}

Код клиента...
const numberOfMales = people.filter(p => p.isMale).length;
```

После выполнения этого рефакторинга все знания о подклассах надежно заключены в суперклассе и функции фабрики. (Обычно я опасаюсь, что суперкласс обращается к подклассу, но этот код не доживет даже до моей следующей чашки чая, поэтому беспокоиться об этом я не буду.)

Теперь я добавляю поле для представления различий между подклассами.

```
class Person...
constructor(name, genderCode) {
  this._name = name;
  this._genderCode = genderCode || "X";
}
get genderCode() {return this._genderCode;}
```

При инициализации я устанавливаю значение по умолчанию. (В качестве примечания — хотя подавляющее большинство людей могут быть классифицированы как мужчины или женщины, в мире становится все больше людей, которые не могут быть отнесены ни к тем, ни к другим. Распространенная ошибка моделирования — забывать об этом.)

Затем я беру случай мужчины и помещаю его логику в суперкласс. Это действие включает модификацию фабрики для возврата `Person` и изменение всех тестов `instanceof` таким образом, чтобы использовалось поле кода пола.

```
function createPerson(aRecord) {
  switch (aRecord.gender) {
    case 'M': return new Person(aRecord.name, "M");
    case 'F': return new Female(aRecord.name);
    default : return new Person(aRecord.name);
  }
}

class Person...
get isMale() {return "M" === this._genderCode;}
```

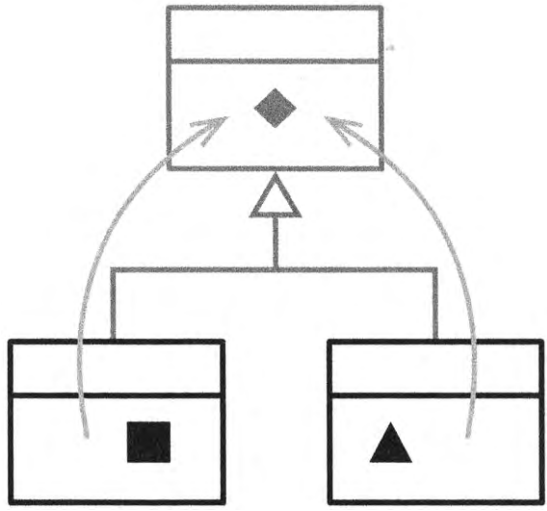
Я выполняю тестирование, удаляю подкласс мужчины, снова провожу тестирование и повторяю те же действия для класса женщины.

```
function createPerson(aRecord) {
  switch (aRecord.gender) {
    case 'M': return new Person(aRecord.name, "M");
    case 'F': return new Person(aRecord.name, "F");
    default : return new Person(aRecord.name);
  }
}
```

Меня раздражает отсутствие симметрии в коде пола. Будущий читатель моего кода всегда будет удивляться этой несимметричности, а потому я предпочитаю изменить код, сделав его симметричным (если я могу сделать это, не внося никакой иной сложности, как в данной ситуации).

```
function createPerson(aRecord) {
  switch (aRecord.gender) {
    case 'M': return new Person(aRecord.name, "M");
    case 'F': return new Person(aRecord.name, "F");
    default : return new Person(aRecord.name, "X");
  }
}

class Person...
constructor(name, genderCode) {
  this._name = name;
  this._genderCode = genderCode ~~|| "X"~~;
}
```

Извлечение суперкласса (Extract Superclass)



```
class Department {
  get totalAnnualCost() {...}
  get name()            {...}
  get headCount()       {...}
}
```

```
class Employee {
  get annualCost() {...}
  get name()       {...}
  get id()         {...}
}
```

```
class Party {
  get name()       {...}
  get annualCost() {...}
}

class Department extends Party {
  get annualCost() {...}
  get headCount()  {...}
}

class Employee extends Party {
  get annualCost() {...}
  get id()         {...}
}
```

Мотивация

Определив, что два класса делают похожие вещи, можно воспользоваться преимуществами механизма наследования, чтобы объединить их сходства в суперкласс. Я могу использовать рефакторинг *Подъем поля* (с. 397) для перенесения в суперкласс общих данных и *Подъем метода* (с. 394) для перенесения туда общего поведения.

Многие авторы по объектно-ориентированному программированию рассматривают наследование как нечто, что должно быть тщательно спланировано заранее, на основе некоторой классификации в "реальном мире". Такая классификация может быть подсказкой к использованию наследования, но не менее часто наследование представляет собой просто то, что я намереваюсь реализовать в ходе развития своей программы, когда нахожу общие элементы, которые хочу объединить.

Альтернативой рефакторингу *Извлечение суперкласса* является рефакторинг *Извлечение класса* (с. 229). По сути, вы выбираете между наследованием или делегированием как способом унификации дублированного поведения. Часто *Извлечение суперкласса* является более простым подходом, поэтому я сделаю его первым, зная, что позже, если потребуется, смогу использовать рефакторинг *Замена суперкласса делегатом* (с. 443).

Техника

- Создайте пустой суперкласс. Сделайте исходные классы его подклассами.

  При необходимости примените к конструкторам рефакторинг *Изменение объявления функции* (с. 170).
- Выполните тестирование.
- Для перемещения общих элементов в суперкласс поочередно используйте рефакторинги *Подъем тела конструктора* (с. 399), *Подъем метода* (с. 394) и *Подъем поля* (с. 397).
- Изучите оставшиеся методы подклассов. Посмотрите, нет ли у них общих частей. Если есть — примените рефакторинг *Извлечение функции* (с. 152), а после него — *Подъем метода* (с. 394).
- Проверьте клиентов исходных классов. Рассмотрите возможность их модификации для использования интерфейса суперкласса.

Пример

Я размышляю над двумя классами, совместно использующими некоторые общие функциональные возможности — имя и годовые и ежемесячные расходы.

```
class Employee {
  constructor(name, id, monthlyCost) {
    this._id = id;
    this._name = name;
    this._monthlyCost = monthlyCost;
  }
  get monthlyCost() {return this._monthlyCost;}
  get name() {return this._name;}
  get id() {return this._id;}

  get annualCost() {
    return this.monthlyCost * 12;
  }
}

class Department {
  constructor(name, staff){
    this._name = name;
    this._staff = staff;
  }
  get staff() {return this._staff.slice();}
  get name() {return this._name;}

  get totalMonthlyCost() {
    return this.staff
```

```
        .map(e => e.monthlyCost)
        .reduce((sum, cost) => sum + cost);
  }
  get headCount() {
    return this.staff.length;
  }
  get totalAnnualCost() {
    return this.totalMonthlyCost * 12;
  }
}
```

Общее поведение можно сделать более явным, извлекая из классов общий суперкласс.

Я начинаю с создания пустого суперкласса и делаю рассматриваемые классы его наследниками.

```
class Party {}

class Employee extends Party {
  constructor(name, id, monthlyCost) {
    super();
    this._id = id;
    this._name = name;
    this._monthlyCost = monthlyCost;
  }
  // Остальная часть класса...

class Department extends Party {
  constructor(name, staff){
    super();
    this._name = name;
    this._staff = staff;
  }
  // Остальная часть класса...
```

Занимаясь рефакторингом *Извлечение суперкласса*, я предпочитаю начинать с данных, что в JavaScript включает работу с конструктором. Так что я начинаю с рефакторинга *Подъем поля* (с. 397), поднимая имя.

```
class Party...
constructor(name){
  this._name = name;
}

class Employee...
constructor(name, id, monthlyCost) {
  super(name);
  this._id = id;
  this._monthlyCost = monthlyCost;
}
```

```
class Department...
constructor(name, staff){
  super(name);
  this._staff = staff;
}
```

Перенося данные в суперкласс, я могу также применить рефакторинг *Подъем метода* (с. 394) к соответствующим методам. В частности, для имени:

```
class Party...
get name() {return this._name;}

class Employee...
get name() {return this._name;}

class Department...
get name() {return this._name;}
```

У меня есть также два метода с похожими телами.

```
class Employee...
get annualCost() {
  return this.monthlyCost * 12;
}
class Department...
get totalAnnualCost() {
  return this.totalMonthlyCost * 12;
}
```

Методы, которые они используют, `monthlyCost` and `totalMonthlyCost`, имеют различные имена и тела — но представляют ли они одно и то же намерение? Если да, я должен использовать рефакторинг *Изменение объявления функции* (с. 170), чтобы унифицировать их имена.

```
class Department...
get totalAnnualCost() {
  return this.monthlyCost * 12;
}

get monthlyCost() { ... }
```

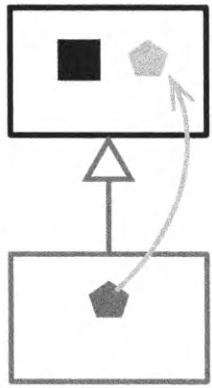
Затем я выполняю переименование, чтобы имена самих методов были одинаковы

```
class Department...
get annualCost() {
  return this.monthlyCost * 12;
}
```

Теперь можно применить рефакторинг *Подъем метода* (с. 394) к методам вычисления годовых расходов.

```
class Party...
get annualCost() {
  return this.monthlyCost * 12;
}

class Employee...
get annualCost() {
  return this.monthlyCost * 12;
}

class Department...
get annualCost() {
  return this.monthlyCost * 12;
}
```

Свертывание иерархии (Collapse Hierarchy)



```
class Employee {...}
class Salesman extends Employee {...}
```

```
class Employee {...}
```

Мотивация

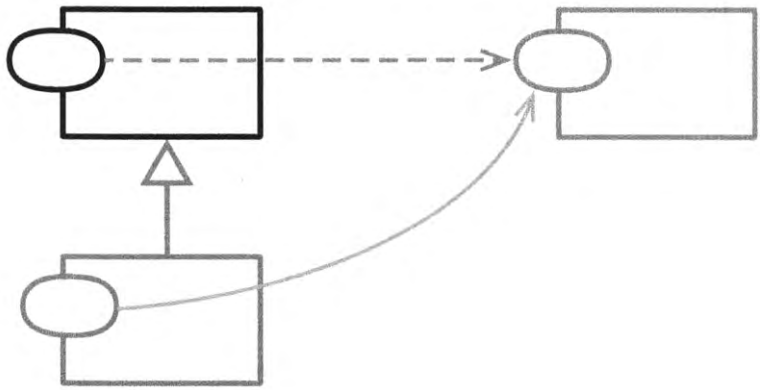
Выполняя рефакторинг иерархии классов, я часто перемещаю методы классов вверх и вниз по иерархии. По мере работы я иногда обнаруживаю, что класс и его родительский элемент уже недостаточно различаются, чтобы их имело смысл разделять. В таком случае я объединяю их вместе.

Техника

- Выберите, какой из классов будет удален.

  Я делаю выбор, исходя из того, какое имя будет иметь больше смысла в будущем. Если ни одно из имен не является наилучшим, мой выбор произвольный.
- Примените рефакторинги *Подъем поля* (с. 397), *Опускание поля* (с. 404), *Подъем метода* (с. 394) и *Опускание метода* (с. 403) для переноса всех элементов в единственный класс.
- Исправьте все обращения к удаляемому классу на обращения к классу, который остается в программе.
- Удалите пустой класс.
- Выполните тестирование.

Замена подкласса делегатом (Replace Subclass with Delegate)



```
class Order {
  get daysToShip() {
    return this._warehouse.daysToShip;
  }
}

class PriorityOrder extends Order {
  get daysToShip() {
    return this._priorityPlan.daysToShip;
  }
}
```

```
class Order {
  get daysToShip() {
    return (this._priorityDelegate)
        ? this._priorityDelegate.daysToShip
        : this._warehouse.daysToShip;
  }
}

class PriorityOrderDelegate {
  get daysToShip() {
    return this._priorityPlan.daysToShip
  }
}
```

Мотивация

Если у меня есть несколько объектов, поведение которых варьируется от категории к категории, наследование является естественным механизмом выражения этого факта. Я помещаю все общие данные и поведение в суперкласс и позволяю каждому подклассу по мере необходимости добавлять и переопределять функции. Объектно-ориентированные языки позволяют реализовать это с помощью простого и знакомого механизма.

Однако наследование имеет и недостатки. Очевидно, что это карта, которую можно разыграть только один раз. Если у меня есть несколько причин для изменения чего-либо, то наследование можно использовать только для одной оси вариации. Так, если я хочу изменять поведение людей в зависимости от их возрастной категории и уровня дохода, то у меня могут быть подклассы для молодых и пожилых, или для состоятельных и бедных — но не одновременно оба варианта.

Еще одна проблема заключается в том, что наследование вводит очень тесную связь между классами. Любое изменение, которое я хочу внести в родительский класс, может легко нарушить работу дочернего класса, поэтому я должен быть очень осторожным и хорошо понимать, как выполняется наследование. Проблема усугубляется, когда логика двух классов находится в разных модулях и поддерживается разными командами разработчиков.

Делегирование решает обе эти проблемы. Я могу выполнять делегирование многим разным классам по разным причинам. Делегирование — это обычные отношения между объектами, так что я могу работать с понятным интерфейсом с гораздо меньшим уровнем связывания, чем при использовании подклассов. Поэтому это обычная практика — встретив проблемы с подклассами, применять рефакторинг *Замена подкласса делегатом*.

Существует популярный принцип: "Пользуйся композицией, а не наследованием" (где композиция по сути представляет собой то же самое, что и делегирование). Многие люди воспринимают его как "наследование — это плохо" и утверждают, что наследование вообще не следует использовать. Я часто прибегаю к наследованию, отчасти потому, что знаю, что в любой момент смогу использовать рефакторинг *Замена подкласса делегатом*, если мне потребуется позже его изменить. Наследование является ценным механизмом, который большую часть времени без проблем выполняет свою работу. Поэтому я начинаю с него и перехожу к делегированию, только когда наследование начинает вызывать проблемы. Такое использование наследования фактически согласуется с принципом из книги [11], который поясняет, как наследование и композиция работают вместе. Этот принцип был реакцией на злоупотребление наследованием.

Тем, кто знаком с книгой [11], будет полезно рассматривать этот рефакторинг как замену подклассов с помощью проектных шаблонов *Состояние* или *Стратегия*. Структурно оба эти шаблона одинаковы. Не во всех случаях рефакторинга *Замена подкласса делегатом* для делегата используется иерархия наследования (показанная в первом примере ниже), но установить иерархию для состояний или стратегий часто бывает полезным.

Техника

- Если имеется много вызовов конструкторов, примените рефакторинг *Замена конструктора фабричной функцией* (с. 379).
- Создайте пустой класс для делегата. Его конструктор должен принимать какие-то специфичные для подкласса данные (а также, как правило, обратную ссылку на суперкласс).
- Добавьте в суперкласс поле для хранения делегата.
- Измените создание подкласса таким образом, чтобы он инициализировал поле делегата экземпляром делегата.

  Это можно сделать в фабричной функции или в конструкторе, если конструктор может надежно указать, следует ли создавать правильный делегат.
- Выберите метод подкласса для переноса в класс делегата.
- Воспользуйтесь рефакторингом *Перенос функции* (с. 244) для его переноса в класс делегата, но не удаляйте при этом исходный делегируемый код.

  Если методу нужны элементы, которые должны быть перенесены в делегат, переместите их. Если ему нужны элементы, которые должны оставаться в суперклассе, добавьте к делегату поле, которое ссылается на суперкласс.

- Если исходный метод вызывается извне класса, переместите исходный делегируемый код из подкласса в суперкласс, защищая его с помощью проверки наличия делегата. Если же нет — примените рефакторинг *Удаление неработающего кода* (с. 283).

  Если имеется несколько подклассов, и вы начинаете дублировать в них код — используйте рефакторинг *Извлечение суперкласса* (с. 418). В этом случае любые делегирующие методы в исходном суперклассе, если поведение по умолчанию перемещено в суперкласс делегата, больше не нуждаются в защите.
- Выполните тестирование.
- Повторяйте эти действия, пока не будут перенесены все методы подкласса.
- Найдите все вызовы конструкторов подклассов и измените их таким образом, чтобы в них использовался конструктор суперкласса.
- Выполните тестирование.
- Примените рефакторинг *Удаление неработающего кода* (с. 283) к подклассу.

Пример

У меня есть класс, который выполняет бронирование на шоу.

```
class Booking...
constructor(show, date) {
  this._show = show;
  this._date = date;
}
```

Имеется подкласс премиум-бронирования, который учитывает различные дополнительные возможности.

```
class PremiumBooking extends Booking...
constructor(show, date, extras) {
  super(show, date);
  this._extras = extras;
}
```

Есть довольно много мелких изменений, которые премиум-бронирование вносит в то, что оно наследует от суперкласса. Как это обычно случается для такого программирования на основании отличий, в некоторых случаях подкласс перекрывает методы суперкласса, в других он добавляет новые методы, которые относятся только к подклассу. Я не буду вдаваться во все из этих вариантов, но выберу несколько интересных случаев.

Во-первых, есть простое перекрытие. Обычное бронирование предлагает двустороннюю связь в прямом эфире после шоу, но только в непиковые дни.

```
class Booking...
get hasTalkback() {
  return this._show.hasOwnProperty('talkback') && !this.isPeakDay;
}
```

Премиум-бронирования перекрывают это поведение, предлагая двустороннюю связь в прямом эфире в любой день.

```
class PremiumBooking...
get hasTalkback() {
  return this._show.hasOwnProperty('talkback');
}
```

Определение цены представляет собой аналогичное переопределение, но в этом случае премиум-метод вызывает метод суперкласса.

```
class Booking...
get basePrice() {
  let result = this._show.price;
  if (this.isPeakDay) result += Math.round(result * 0.15);
  return result;
}

class PremiumBooking...
get basePrice() {
  return Math.round(super.basePrice + this._extras.premiumFee);
}
```

Последний пример — когда премиум-бронирование предлагает поведение, которого нет в суперклассе.

```
class PremiumBooking...
get hasDinner() {
  return this._extras.hasOwnProperty('dinner') && !this.isPeakDay;
}
```

Для этого примера хорошо работает наследование. Я могу разобраться в базовом классе без необходимости понимания подкласса. Подкласс определен просто указанием его отличий от базового варианта — что сокращает количество дублирования и ясно говорит о том, какие отличия от базового класса он вводит.

Конечно, все не так идеально, как говорится в предыдущем абзаце. В структуре суперкласса есть вещи, которые имеют смысл только благодаря подклассу, — например методы, которые были разложены так, чтобы упростить перекрытие только нужных фрагментов поведения. Таким образом, хотя большую часть времени я могу изменять базовый класс без необходимости понимать подклассы, бывают ситуации, когда такое сознательное игнорирование подклассов может привести к нарушению их работоспособности при модификации суперкласса. Впрочем, если такого рода ситуации редки, то наследование окупается (при условии, что у меня есть хорошие тесты для обнаружения неприятностей в подклассе).

Так зачем мне изменять такую хорошую ситуацию, используя рефакторинг *Замена подкласса делегатом*? Наследование — это инструмент, который можно использовать только один раз, так что если у меня есть иная причина использовать наследование, и я полагаю, что здесь оно принесет мне больше пользы, чем подкласс премиум-бронирования, то обрабатывать премиум-бронирование лучше другим способом. Кроме того, мне может понадобиться изменить бронирование по умолчанию на премиум-бронирование динамически, т.е. поддерживать такой метод, как `aBooking.bePremium()`. В некоторых случаях я могу избежать этого, создав совершенно новый объект (распространенным примером является HTTP-запрос, который загружает с сервера новые данные). Но иногда мне нужно изменить структуру данных, а не перестраивать ее с нуля. Сложной задачей оказывается и простая замена единственного бронирования, на которое имеются ссылки из разных мест. В таких ситуациях может быть полезно разрешить бронированию переключаться со стандартного на премиум и обратно.

Когда возникают такие потребности, мне нужно применить рефакторинг *Замена подкласса делегатом*. Для выполнения бронирования у меня есть клиентские вызовы конструкторов двух классов:

```
Обычное бронирование
aBooking = new Booking(show,date);

Премиум-бронирование
aBooking = new PremiumBooking(show, date, extras);
```

Удаление подклассов изменит все эту ситуацию, так что я предпочитаю инкапсулировать вызовы конструктора с помощью рефакторинга *Замена конструктора фабричной функцией* (с. 379).

```
Верхний уровень...
function createBooking(show, date) {
  return new Booking(show, date);
}

function createPremiumBooking(show, date, extras) {
  return new PremiumBooking (show, date, extras);
}

Обычное бронирование
aBooking = createBooking(show, date);

Премиум-бронирование
aBooking = createPremiumBooking(show, date, extras);
```

Теперь я создаю новый класс делегата. Параметрами его конструктора являются те параметры, которые используются только в подклассе, а также обратная ссылка на бронирующий объект. Она нужна потому, что несколько методов

подкласса требуют доступа к данным, хранящимся в суперклассе. Наследование позволяет легко этого добиться, но при использовании делегата необходима обратная ссылка.

```
class PremiumBookingDelegate...
constructor(hostBooking, extras) {
  this._host = hostBooking;
  this._extras = extras;
}
```

Теперь я подключаю нового делегата к объекту бронирования. Я делаю это, изменяя фабричную функцию для премиум-бронирования.

```
Верхний уровень...
function createPremiumBooking(show, date, extras) {
  const result = new PremiumBooking (show, date, extras);
  result._bePremium(extras);
return result;
}

class Booking...
_bePremium(extras) {
  this._premiumDelegate = new PremiumBookingDelegate(this, extras);
}
```

В _bePremium использовано подчеркивание, чтобы показать, что этот метод не должен быть частью общедоступного интерфейса Booking. Конечно, если цель этого рефакторинга состоит в том, чтобы позволить преобразовывать бронирование в премиум-бронирование, данный метод может быть открытым.

Кроме того, я могу выполнить все подключения в конструкторе Booking. Для этого мне нужен некоторый способ сообщить конструктору, что у нас есть премиум-бронирование. Это может быть дополнительный параметр или просто использование extras, если я могу быть уверен, что данный параметр всегда присутствует при использовании премиум-бронирования. Здесь я предпочитаю явное указание этого факта путем выполнения фабричной функции.

После того как структуры созданы и настроены, наступает время переноса поведения. Первый случай, который я рассмотрю, — простое перекрытие hasTalkback. Вот имеющийся код:

```
class Booking...
get hasTalkback() {
  return this._show.hasOwnProperty('talkback') && !this.isPeakDay;
}

class PremiumBooking...
get hasTalkback() {
  return this._show.hasOwnProperty('talkback');
}
```

Я использую рефакторинг *Перенос функции* (с. 244) для перемещения метода подкласса в делегат и перенаправляю все обращения к данным суперкласса с помощью вызовов _host.

```
class PremiumBookingDelegate...
get hasTalkback() {
  return this._host._show.hasOwnProperty('talkback');
}

class PremiumBooking...
get hasTalkback() {
  return this._premiumDelegate.hasTalkback;
}
```

Я выполняю тестирование, чтобы убедиться, что все нормально работает, а затем удаляю метод подкласса:

```
class PremiumBooking...
get hasTalkback() {
  return this._premiumDelegate.hasTalkback;
}
```

В этот момент я выполняю тесты, ожидая, что некоторые из них не пройдут.

Перемещение завершается добавлением логики диспетчеризации в метод суперкласса, которая использует делегат в случае его наличия.

```
class Booking...
get hasTalkback() {
  return (this._premiumDelegate)
    ? this._premiumDelegate.hasTalkback
    : this._show.hasOwnProperty('talkback') && !this.isPeakDay;
}
```

Следующий рассматриваемый мною случай — это базовая цена.

```
class Booking...
get basePrice() {
  let result = this._show.price;
  if (this.isPeakDay) result += Math.round(result * 0.15);
  return result;
}

class PremiumBooking...
get basePrice() {
  return Math.round(super.basePrice + this._extras.premiumFee);
}
```

Это почти то же самое, но есть загвоздка в виде надоедливого вызова super (который довольно часто встречается в подобных случаях расширения подклассов). Перемещая код подкласса в делегат, нужно вызвать родительский код, но я не могу просто написать this._host._basePrice, не попав при этом в бесконечную рекурсию.

Есть несколько вариантов решения проблемы. Одним из них является применение рефакторинга *Извлечение функции* (с. 152) к базовому вычислению, чтобы отделить логику диспетчеризации от расчета цены. (Остальная часть перемещения та же, что и ранее.)

```
class Booking...
get basePrice() {
  return (this._premiumDelegate)
    ? this._premiumDelegate.basePrice
    : this._privateBasePrice;
}

get _privateBasePrice() {
  let result = this._show.price;
  if (this.isPeakDay) result += Math.round(result * 0.15);
  return result;
}

class PremiumBookingDelegate...
get basePrice() {
  return Math.round(this._host._privateBasePrice +
                    this._extras.premiumFee);
}
```

В качестве альтернативы можно изменить метод делегата как расширение базового метода.

```
class Booking...
get basePrice() {
  let result = this._show.price;
  if (this.isPeakDay) result += Math.round(result * 0.15);
  return (this._premiumDelegate)
    ? this._premiumDelegate.extendBasePrice(result)
    : result;
}

class PremiumBookingDelegate...
extendBasePrice(base) {
  return Math.round(base + this._extras.premiumFee);
}
```

Оба варианта разумны и работоспособны; я предпочитаю последний, поскольку он немного короче.

Последний случай — это метод, который существует только в подклассе.

```
class PremiumBooking...
get hasDinner() {
  return this._extras.hasOwnProperty('dinner') && !this.isPeakDay;
}
```

Я переношу его из подкласса в делегат:

```
class PremiumBookingDelegate...
get hasDinner() {
  return this._extras.hasOwnProperty('dinner')
     && !this._host.isPeakDay;
}
```

Затем добавляю логику диспетчеризации в Booking:

```
class Booking...
get hasDinner() {
  return (this._premiumDelegate)
    ? this._premiumDelegate.hasDinner
    : undefined;
}
```

В JavaScript доступ к свойству объекта, который не определен, возвращает undefined, поэтому я поступаю здесь именно таким образом. (Хотя все мои инстинкты кричат о том, что нужно заставить метод генерировать ошибку, как было бы в случае других объектно-ориентированных динамических языков, к которым я привык.)

Переместив все поведение из подкласса, можно изменить фабричный метод так, чтобы он возвращал суперкласс; выполнив тестирование и убедившись, что все в порядке, я удалю подкласс.

```
Верхний уровень...
function createPremiumBooking(show, date, extras) {
  const result = new ~~Premium~~Booking (show, date, extras);
  result._bePremium(extras);
  return result;
}

~~class PremiumBooking extends Booking ...~~
```

Это один из тех рефакторингов, где я не чувствую, что рефакторинг улучшает код сам по себе. Наследование справляется с этой ситуацией очень хорошо, тогда как использование делегирования предполагает добавление логики диспетчеризации, двусторонних ссылок, а следовательно, дополнительной сложности. Но этот рефакторинг может все же иметь смысл, поскольку преимущество изменяемого премиум-статуса или необходимость использования наследования для других целей может превалировать над недостатками потери наследования.

Пример: замена иерархии

В предыдущем примере было показано применение рефакторинга *Замена подкласса делегатом* для одного подкласса, но я могу сделать то же самое со всей иерархией.

```
function createBird(data) {
  switch (data.type) {
    case 'EuropeanSwallow':
      return new EuropeanSwallow(data);
    case 'AfricanSwallow':
      return new AfricanSwallow(data);
    case 'NorweigianBlueParrot':
      return new NorwegianBlueParrot(data);
    default:
      return new Bird(data);
  }
}

class Bird {
  constructor(data) {
    this._name = data.name;
    this._plumage = data.plumage;
  }

  get name() {return this._name;}

  get plumage() {
    return this._plumage || "average";
  }

  get airSpeedVelocity() {return null;}
}

class EuropeanSwallow extends Bird {
  get airSpeedVelocity() {return 35;}
}

class AfricanSwallow extends Bird {
  constructor(data) {
    super (data);
    this._numberOfCoconuts = data.numberOfCoconuts;
  }

  get airSpeedVelocity() {
    return 40 - 2 * this._numberOfCoconuts;
  }
}

class NorwegianBlueParrot extends Bird {
  constructor(data) {
    super (data);
    this._voltage = data.voltage;
    this._isNailed = data.isNailed;
  }
```

```
  get plumage() {
    if (this._voltage > 100) return "scorched";
    else return this._plumage || "beautiful";
  }

  get airSpeedVelocity() {
    return (this._isNailed) ? 0 : 10 + this._voltage / 10;
  }
}
```

В скором времени система будет весьма сильно отличаться для птиц в дикой природе и для птиц в неволе. Эта разница может быть смоделирована в виде двух подклассов `Bird`: `WildBird` и `CaptiveBird`. Однако я могу использовать наследование только однократно, а потому, если я хочу использовать подклассы для птиц в природе и неволе, мне придется удалить их для различных видов.

Когда задействовано несколько подклассов, я занимаюсь ими по одному, начиная с простейшего — в данном случае это `EuropeanSwallow`. Я создаю пустой класс делегата.

```
class EuropeanSwallowDelegate {
}
```

Я пока не добавляю никаких данных или обратную ссылку в качестве параметров. В данном примере я буду предоставлять их по мере необходимости.

Мне нужно решить, где именно обрабатывать инициализацию поля делегата. В данном случае, поскольку в одном аргументе данных конструктора у меня есть вся нужная информация, я решил делать это в конструкторе. Поскольку существует несколько делегатов, которые я мог бы добавить, я создаю функцию для выбора правильного делегата на основе кода типа в документе.

```
class Bird...
constructor(data) {
  this._name = data.name;
  this._plumage = data.plumage;
  this._speciesDelegate = this.selectSpeciesDelegate(data);
}

selectSpeciesDelegate(data) {
  switch(data.type) {
    case 'EuropeanSwallow':
      return new EuropeanSwallowDelegate();
    default: return null;
  }
}
```

Теперь, когда я настроил структуру, можно применить рефакторинг *Перенос функции* (с. 244) к скорости полета европейской ласточки.

```
class EuropeanSwallowDelegate...
  get airSpeedVelocity() {return 35;}

class EuropeanSwallow...
get airSpeedVelocity() {
  return this._speciesDelegate.airSpeedVelocity;
}
```

Я изменяю `airSpeedVelocity` в суперклассе так, чтобы вызывался делегат (если таковой имеется).

```
class Bird...
get airSpeedVelocity() {
  return this._speciesDelegate
    ? this._speciesDelegate.airSpeedVelocity
    : null;
}
```

Я удаляю подкласс.

```
class EuropeanSwallow extends Bird {
  get airSpeedVelocity() {
    return this._speciesDelegate.airSpeedVelocity;
  }
}

Верхний уровень...
function createBird(data) {
  switch (data.type) {
    case 'EuropeanSwallow':
      return new EuropeanSwallow(data);
    case 'AfricanSwallow':
      return new AfricanSwallow(data);
    case 'NorweigianBlueParrot':
      return new NorwegianBlueParrot(data);
    default:
      return new Bird(data);
  }
}
```

Далее я занимаюсь африканской ласточкой. Создаю класс; на этот раз конструктору нужны данные.

```
class AfricanSwallowDelegate...
constructor(data) {
  this._numberOfCoconuts = data.numberOfCoconuts;
}

class Bird...
selectSpeciesDelegate(data) {
  switch(data.type) {
    case 'EuropeanSwallow':
```

```
      return new EuropeanSwallowDelegate();
    case 'AfricanSwallow':
      return new AfricanSwallowDelegate(data);
    default: return null;
  }
}
```

Применяю рефакторинг *Перенос функции* (с. 244) к `airSpeedVelocity`.

```
class AfricanSwallowDelegate...
get airSpeedVelocity() {
  return 40 - 2 * this._numberOfCoconuts;
}

class AfricanSwallow...
get airSpeedVelocity() {
  return this._speciesDelegate.airSpeedVelocity;
}
```

Теперь можно удалить подкласс африканской ласточки.

```
class AfricanSwallow extends Bird {
  // Все тело класса ...
}

function createBird(data) {
  switch (data.type) {
    case 'AfricanSwallow':
      return new AfricanSwallow(data);
    case 'NorweigianBlueParrot':
      return new NorwegianBlueParrot(data);
    default:
      return new Bird(data);
  }
}
```

Теперь о норвежском голубом попугае[1]. Для создания класса и переноса скорости полета используются те же шаги, что и раньше, поэтому я просто покажу окончательный результат.

```
class Bird...
selectSpeciesDelegate(data) {
  switch(data.type) {
    case 'EuropeanSwallow':
      return new EuropeanSwallowDelegate();
    case 'AfricanSwallow':
      return new AfricanSwallowDelegate(data);
    case 'NorweigianBlueParrot':
      return new NorwegianBlueParrotDelegate(data);
    default: return null;
  }
}
```

[1] Персонаж скетча из 8-го эпизода "Летающего цирка Монти Пайтона". — *Примеч. пер.*

```
class NorwegianBlueParrotDelegate...
constructor(data) {
  this._voltage = data.voltage;
  this._isNailed = data.isNailed;
}
get airSpeedVelocity() {
  return (this._isNailed) ? 0 : 10 + this._voltage / 10;
}
```

Все хорошо, но норвежский попугай переопределяет свойство оперения `plumage`, с которым мне не приходилось сталкиваться в других случаях. Применение рефакторинга *Перенос функции* (с. 244) оказывается достаточно простым, хотя и с необходимостью изменить конструктор, чтобы добавить обратную ссылку на птицу.

```
class NorwegianBlueParrot...
get plumage() {
  return this._speciesDelegate.plumage;
}

class NorwegianBlueParrotDelegate...
get plumage() {
  if (this._voltage > 100) return "scorched";
  else return this._bird._plumage || "beautiful";
}

constructor(data, bird) {
  this._bird = bird;
  this._voltage = data.voltage;
  this._isNailed = data.isNailed;
}

class Bird...
selectSpeciesDelegate(data) {
  switch(data.type) {
    case 'EuropeanSwallow':
      return new EuropeanSwallowDelegate();
    case 'AfricanSwallow':
      return new AfricanSwallowDelegate(data);
    case 'NorweigianBlueParrot':
      return new NorwegianBlueParrotDelegate(data, this);
  default: return null;
  }
}
```

Сложность заключается в удалении метода подкласса `plumage`. Если я напишу

```
class Bird...
get plumage() {
  if (this._speciesDelegate)
```

```
    return this._speciesDelegate.plumage;
  else
    return this._plumage || "average";
}
```

то получу кучу ошибок, потому что в классах делегатов других видов нет свойства оперения.

Я мог бы использовать более точное условие:

```
class Bird...
get plumage() {
  if (this._speciesDelegate instanceof NorwegianBlueParrotDelegate)
    return this._speciesDelegate.plumage;
  else
    return this._plumage || "average";
}
```

Но я надеюсь, что этот код пахнет для вас таким же разлагающимся попугаем, как и для меня. Практически никогда не стоит использовать явную проверку классов, подобную показанной.

Другой вариант — реализовать случай по умолчанию для других делегатов.

```
class Bird...
get plumage() {
  if (this._speciesDelegate)
    return this._speciesDelegate.plumage;
  else
    return this._plumage || "average";
}

class EuropeanSwallowDelegate...
get plumage() {
  return this._bird._plumage || "average";
}

class AfricanSwallowDelegate...
get plumage() {
  return this._bird._plumage || "average";
}
```

Но при этом получается дублирование метода по умолчанию для оперения. И если это недостаточно плохо, то учтите, что я, кроме того, получаю бонус — дублирование в конструкторах для присваивания обратной ссылки.

Решением проблемы дублирования, естественно, является наследование — я применяю рефакторинг *Извлечение суперкласса* (с. 418) к делегатам видов.

```
class SpeciesDelegate {
  constructor(data, bird) {
    this._bird = bird;
  }
```

```
  get plumage() {
    return this._bird._plumage || "average";
  }

class EuropeanSwallowDelegate extends SpeciesDelegate {

class AfricanSwallowDelegate extends SpeciesDelegate {
  constructor(data, bird) {
    super(data,bird);
    this._numberOfCoconuts = data.numberOfCoconuts;
  }

class NorwegianBlueParrotDelegate extends SpeciesDelegate {
  constructor(data, bird) {
    super(data, bird);
    this._voltage = data.voltage;
    this._isNailed = data.isNailed;
  }
```

Теперь у меня есть суперкласс, и я могу переместить любое поведение по умолчанию из `Bird` в `SpeciesDelegate`, гарантируя, что в поле `speciesDelegate` всегда что-то есть.

```
class Bird...
selectSpeciesDelegate(data) {
  switch(data.type) {
    case 'EuropeanSwallow':
      return new EuropeanSwallowDelegate(data, this);
    case 'AfricanSwallow':
      return new AfricanSwallowDelegate(data, this);
    case 'NorweigianBlueParrot':
      return new NorwegianBlueParrotDelegate(data, this);
    default: return new SpeciesDelegate(data, this);
  }
}

// Остальная часть кода Bird...

get plumage() {return this._speciesDelegate.plumage;}
get airSpeedVelocity() {
  return this._speciesDelegate.airSpeedVelocity;
}

class SpeciesDelegate...
get airSpeedVelocity() {return null;}
```

Мне нравится полученный результат, так как он упрощает методы делегирования в `Bird`. Я легко вижу, какое поведение делегируется делегату видов. Вот окончательное состояние этих классов.

```
function createBird(data) {
  return new Bird(data);
}

class Bird {
  constructor(data) {
    this._name = data.name;
    this._plumage = data.plumage;
    this._speciesDelegate = this.selectSpeciesDelegate(data);
  }
  get name() {return this._name;}
  get plumage() {return this._speciesDelegate.plumage;}
  get airSpeedVelocity() {
    return this._speciesDelegate.airSpeedVelocity;
  }

  selectSpeciesDelegate(data) {
    switch(data.type) {
      case 'EuropeanSwallow':
        return new EuropeanSwallowDelegate(data, this);
      case 'AfricanSwallow':
        return new AfricanSwallowDelegate(data, this);
      case 'NorweigianBlueParrot':
        return new NorwegianBlueParrotDelegate(data, this);
      default: return new SpeciesDelegate(data, this);
    }
  }
  // Остальная часть кода Bird...
}

class SpeciesDelegate {
  constructor(data, bird) {
    this._bird = bird;
  }
  get plumage() {
    return this._bird._plumage || "average";
  }
  get airSpeedVelocity() {return null;}
}

class EuropeanSwallowDelegate extends SpeciesDelegate {
  get airSpeedVelocity() {return 35;}
}

class AfricanSwallowDelegate extends SpeciesDelegate {
  constructor(data, bird) {
    super(data,bird);
    this._numberOfCoconuts = data.numberOfCoconuts;
  }
```

```
  get airSpeedVelocity() {
    return 40 - 2 * this._numberOfCoconuts;
  }
}

class NorwegianBlueParrotDelegate extends SpeciesDelegate {
  constructor(data, bird) {
    super(data, bird);
    this._voltage = data.voltage;
    this._isNailed = data.isNailed;
  }
  get airSpeedVelocity() {
    return (this._isNailed) ? 0 : 10 + this._voltage / 10;
  }
  get plumage() {
    if (this._voltage > 100) return "scorched";
    else return this._bird._plumage || "beautiful";
  }
}
```

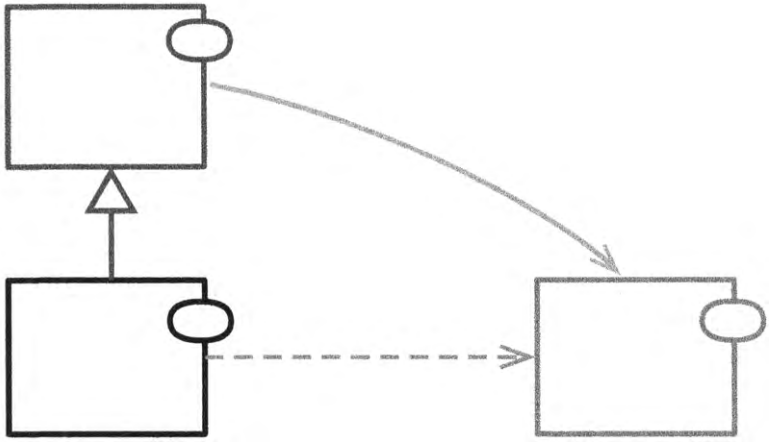
В этом примере исходные подклассы заменены делегатом, но в `SpeciesDelegate` все еще имеется очень похожая структура наследования. Получил ли я что-нибудь от этого рефакторинга, кроме освобождения `Bird` от наследования? Наследование видов теперь более ограничено и охватывает только данные и функции, которые варьируются в зависимости от вида. Любой код, одинаковый для всех видов, остается в `Bird` и его будущих подклассах.

Я мог бы применить ту же идею создания делегата суперкласса к рассматривавшемуся ранее примеру бронирования. Это позволило бы мне заменить те методы в `Booking`, которые содержат логику диспетчеризации, простыми вызовами делегата и обеспечить обработку диспетчеризации его наследованием. Однако мы и так потратили немало времени, поэтому я оставлю это решение в качестве упражнения для читателя.

Данные примеры иллюстрируют, что принцип "Пользуйся композицией, а не наследованием" лучше было бы сформулировать как "Подумайте о разумном объединении композиции и наследования вместо использования чего-то одного", но, боюсь, это звучит не так броско.

Замена суперкласса делегатом (Replace Superclass with Delegate)

Бывший рефакторинг *Замена наследования делегированием*



```
class List {...}
class Stack extends List {...}
```

```
class Stack {
  constructor() {
    this._storage = new List();
  }
}
class List {...}
```

Мотивация

В объектно-ориентированных программах наследование является мощным и легко доступным способом повторного использования существующей функциональности. Я наследую некоторый существующий класс, затем выполняю перекрытие и добавление новой функциональности. Но наследование может быть выполнено таким образом, что приводит к путанице и сложностям.

Одним из классических примеров неправильного наследования с первых дней существования объектов была трактовка стека как подкласса списка. К этому привела идея, состоявшая в повторном использовании хранилища данных списка и операций для управления им. Несмотря на то что повторное использование

в общем случае вещь хорошая, у данного наследования была большая проблема: в интерфейсе стека присутствовали все операции списка, хотя большинство из них к стеку были не применимы. Куда лучший подход состоит в том, чтобы превратить список в поле стека и делегировать ему необходимые операции.

Это пример одной из причин использования рассматриваемого рефакторинга *Замена суперкласса делегатом* — если функции суперкласса не имеют смысла для подкласса, это признак того, что я не должен применять наследование для использования функциональности суперкласса.

Для использования наследования помимо применимости всех функций суперкласса также должно быть верно и то, что каждый экземпляр подкласса является экземпляром суперкласса и допустимым объектом везде, где мы используем суперкласс. Если у меня есть класс модели автомобиля с такой информацией, как название и объем двигателя, я мог бы решить, что смогу повторно использовать эти функции для представления физического автомобиля, добавляя функции регистрационного номера и даты производства. Это распространенная (и часто трудно уловимая) ошибка моделирования, которую я называю омонимом экземпляра типа [37].

Это два примера проблем, приводящих к путанице и ошибкам, которых можно легко избежать, заменив наследование делегированием отдельному объекту. Использование делегирования проясняет, что это отдельная сущность, в которой используются только некоторые ее функции.

Даже в тех случаях, когда подкласс является разумной моделью, я могу использовать рефакторинг *Замена суперкласса делегатом*, потому что взаимоотношения между подклассом и суперклассом являются тесной связью, при которой работоспособность подкласса может быть легко нарушена внесением изменений в суперкласс. Недостатком делегирования является то, что мне нужно писать передающую функцию для любой функции, которая работает и в основном классе, и в делегате. И пусть писать такие функции очень скучно, по счастью, они слишком просты, чтобы можно было ошибиться при их написании.

Как следствие, некоторые люди советуют полностью избегать наследования, но я не согласен с таким категорическим решением. При условии выполнения соответствующих семантических условий (каждый метод супертипа применим к подтипу, каждый экземпляр подтипа является экземпляром супертипа), наследование является простым и эффективным механизмом. Я могу легко применить рефакторинг *Замена суперкласса делегатом*, если позже ситуация изменится и наследование больше не будет лучшим вариантом. Поэтому (как правило) я советую сначала использовать наследование, а когда (и если) оно вызовет проблемы — прибегнуть к расматриваемому в этом разделе рефакторингу.

Техника

- Создайте в подклассе поле, которое ссылается на объект суперкласса. Инициализируйте эту ссылку делегата новым экземпляром.
- Для каждого элемента суперкласса создайте в подклассе функцию передачи, которая выполняет передачу созданной ссылке на делегата. Выполните тестирование после реализации передачи для каждой согласованной группы.

  В большинстве случаев можно выполнять тестирование после осуществления передачи для каждой очередной функции, но, например, пары `get/set` могут быть протестированы только вместе, после того как обе функции были перемещены.
- Когда все элементы суперкласса окажутся перекрыты передающими функциями, удалите наследование.

Пример

Недавно я консультировал библиотеку древних свитков в одном старом городе. Они хранят информацию о своих свитках в каталоге. Каждый свиток имеет идентификационный номер, заголовок и список тематических меток (тегов).

```
class CatalogItem...
constructor(id, title, tags) {
  this._id = id;
  this._title = title;
  this._tags = tags;
}

get id() {return this._id;}
get title() {return this._title;}
hasTag(arg) {return this._tags.includes(arg);}
```

Свиткам требуется регулярная очистка. Код для этого использует элемент каталога и расширяет его данными, необходимыми для очистки.

```
class Scroll extends CatalogItem...
constructor(id, title, tags, dateLastCleaned) {
  super(id, title, tags);
  this._lastCleaned = dateLastCleaned;
}

needsCleaning(targetDate) {
  const threshold = this.hasTag("revered") ? 700 : 1500;
  return this.daysSinceLastCleaning(targetDate) > threshold ;
}

daysSinceLastCleaning(targetDate) {
  return this._lastCleaned.until(targetDate, ChronoUnit.DAYS);
}
```

Здесь мы имеем дело с примером распространенной ошибки моделирования. Существует разница между физическим свитком и элементом каталога. Свиток, описывающий лечение от бубонной чумы, может иметь несколько копий, но быть единственным элементом в каталоге.

Во многих ситуациях я могу избежать подобной ошибки. Я могу рассматривать заголовок и теги как копии данных в каталоге. Если эти данные никогда не изменяются, это может успешно пройти. Но при необходимости обновления придется быть очень осторожным и убедиться, что все копии одного и того же элемента каталога корректно обновлены.

Даже без учета этой проблемы я все равно хотел бы изменить взаимоотношения между классами. Использование элемента каталога в качестве суперкласса для свитка в будущем может запутать программистов и поэтому является плохой моделью для работы.

Я начинаю с создания в `Scroll` свойства, которое ссылается на элемент каталога и инициализируется новым экземпляром.

```
class Scroll extends CatalogItem...
constructor(id, title, tags, dateLastCleaned) {
  super(id, title, tags);
  this._catalogItem = new CatalogItem(id, title, tags);
  this._lastCleaned = dateLastCleaned;
}
```

Я создаю методы передачи для каждого элемента суперкласса, который используется в подклассе.

```
class Scroll...
get id() {return this._catalogItem.id;}
get title() {return this._catalogItem.title;}
hasTag(aString) {return this._catalogItem.hasTag(aString);}
```

Теперь я удаляю наследование элемента каталога.

```
class Scroll ~~extends CatalogItem~~ {
  constructor(id, title, tags, dateLastCleaned) {
  ~~super(id, title, tags);~~
  this._catalogItem = new CatalogItem(id, title, tags);
  this._lastCleaned = dateLastCleaned;
}
```

Удаление наследования завершает основную часть рефакторинга *Замена суперкласса делегатом*, но в этом случае нужно сделать кое-что еще.

Рефакторинг меняет роль элемента каталога на роль компонента свитка; каждый свиток теперь содержит уникальный экземпляр элемента каталога. Во многих случаях при выполнении данного рефакторинга этого достаточно. Однако в нашей ситуации лучшей моделью является привязка элемента каталога к свиткам-копиям в библиотеке. Это, по сути, означает применение рефакторинга *Замена значения ссылкой* (с. 300).

Однако есть проблема, которую следует решить, прежде чем использовать указанный рефакторинг. В исходной структуре наследования свиток использовал для хранения идентификатора поле идентификатора элемента каталога. Но если я рассматриваю элемент каталога как ссылку, он должен использовать этот идентификатор в качестве идентификатора элемента каталога, а не идентификатора свитка. Это означает, что мне нужно создать поле идентификатора в свитке и использовать его вместо идентификатора в элементе каталога. Это частично перемещение, частично расщепление.

```
class Scroll...
constructor(id, title, tags, dateLastCleaned) {
  this._id = id;
  this._catalogItem = new CatalogItem(null, title, tags);
  this._lastCleaned = dateLastCleaned;
}

get id() {return this._id;}
```

Создание элемента каталога с нулевым идентификатором обычно вызывает срабатывание тревожной сигнализации и выезда опергруппы. Но это временное явление; когда я завершу работу, свитки будут ссылаться на общий элемент каталога с надлежащим идентификатором.

В настоящее время свитки загружаются как часть процедуры загрузки.

```
Процедура загрузки...
const scrolls = aDocument
  .map(record => new Scroll(record.id,
                            record.catalogData.title,
                            record.catalogData.tags,
                            LocalDate.parse(record.lastCleaned)));
```

Первый шаг рефакторинга *Замена значения ссылкой* (с. 300) состоит в поиске или создании хранилища. Я обнаружил, что существует хранилище, которое я могу легко импортировать в подпрограмму загрузки. Хранилище предоставляет элементы каталога, индексированные идентификатором. Моя следующая задача — увидеть, как получить этот идентификатор в конструктор свитка. К счастью, он присутствует во входных данных и игнорируется, так как при использовании наследования он был бесполезен. После этого я могу использовать рефакторинг *Изменение объявления функции* (с. 170), чтобы добавить каталог и идентификатор элемента каталога в параметры конструктора.

```
Процедура загрузки...
const scrolls = aDocument
  .map(record => new Scroll(record.id,
                            record.catalogData.title,
                            record.catalogData.tags,
                            LocalDate.parse(record.lastCleaned),
```

```
                            record.catalogData.id,
                            catalog));

class Scroll...
constructor(id, title, tags, dateLastCleaned,
            catalogID, catalog) {
  this._id = id;
  this._catalogItem = new CatalogItem(null, title, tags);
  this._lastCleaned = dateLastCleaned;
}
```

Теперь я модифицирую конструктор так, чтобы использовать идентификатор каталога для поиска элемента каталога и применить его вместо создания нового элемента.

```
class Scroll...
constructor(id, title, tags, dateLastCleaned, catalogID, catalog) {
  this._id = id;
  this._catalogItem = catalog.get(catalogID);
  this._lastCleaned = dateLastCleaned;
}
```

Мне больше не нужны заголовок и теги, переданные в конструктор, поэтому я использую рефакторинг *Изменение объявления функции* (с. 170) для их удаления.

```
Процедура загрузки...
const scrolls = aDocument
  .map(record => new Scroll(record.id,
                            record.catalogData.title,
                            record.catalogData.tags,
                            LocalDate.parse(record.lastCleaned),
                            record.catalogData.id,
                            catalog));
class Scroll...
constructor(id, title, tags, dateLastCleaned,
            catalogID, catalog) {
  this._id = id;
  this._catalogItem = catalog.get(catalogID);
  this._lastCleaned = dateLastCleaned;
}
```

Библиография

Вы можете найти онлайн-версию этой библиографии по адресу `https://martinfowler.com/books/refactoring-bibliography.html`. Многие из записей в этой библиографии относятся к "bliki" — разделу `martinfowler.com`, где я предоставляю краткие описания различных терминов, используемых при разработке программного обеспечения. При написании этой книги я решил не включать объяснения в текст книги, указав читателям, где их найти.

1. Scott W. Ambler and Pramod J. Sadalage. *Refactoring Databases.* Addison-Wesley, 2006. ISBN 0321293533. (Скотт В. Эмблер, Прамодкумар Дж. Садаладж, Рефакторинг баз данных: эволюционное проектирование, пер. с англ., ИД "Вильямс", 2007 г.)
2. `https://babeljs.io`.
3. Jay Bazuzi. "Safely Extract a Method in Any C++ Code". `http://jay.bazuzi.com/Safely-extract-a-method-in-any-C++-code/`.
4. Kent Beck. *Smalltalk Best Practice Patterns.* Addison-Wesley, 1997. ISBN 013476904X.
5. `http://chaijs.com`.
6. `http://www.eclipse.org`.
7. Michael Feathers. *Working Effectively with Legacy Code.* Prentice Hall, 2004. ISBN 0131177052. (Майкл Физерс, Эффективная работа с унаследованным кодом, пер. с англ., ИД "Вильямс", 2009 г.)
8. Jay Fields, Shane Harvie, and Martin Fowler. *Refactoring Ruby Edition.* Addison-Wesley, 2009. ISBN 0321603508.
9. Neal Ford, Rebecca Parsons, and Patrick Kua. *Building Evolutionary Architectures.* O'Reilly, 2017. ISBN 1491986360.
10. Nicole Forsgren, Jez Humble, and Gene Kim. *Accelerate: The Science of Lean Software and DevOps: Building and Scaling High Performing Technology Organizations.* IT Revolution Press, 2018. ISBN 1942788339.

11. Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software.* Addison-Wesley, 1994. ISBN 0201634988.

12. Elliotte Rusty Harold. *Refactoring HTML.* Addison-Wesley, 2008. ISBN 0321503635.

13. `https://www.jetbrains.com/idea/`.

14. Joshua Kerievsky. *Refactoring to Patterns.* Addison-Wesley, 2004. ISBN 0321213351. (Джошуа Кериевски, Рефакторинг с использованием шаблонов, пер. с англ., ИД "Вильямс", 2006 г.)

15. `https://langserver.org`.

16. `https://en.wikipedia.org/wiki/Unibroue`.

17. Martin Fowler. "Bliki: TwoHardThings". `https://martinfowler.com/bliki/TwoHardThings.html`.

18. Martin Fowler. "Bliki: BranchByAbstraction". `https://martinfowler.com/bliki/BranchByAbstraction.html`.

19. Martin Fowler. "Collection Pipeline". `https://martinfowler.com/articles/collection-pipeline/`.

20. Martin Fowler. "Bliki: CommandQuerySeparation". `https://martinfowler.com/bliki/CommandQuerySeparation.html`.

21. Martin Fowler. "Bliki: ClockWrapper". `https://martinfowler.com/bliki/ClockWrapper.html`.

22. Martin Fowler. "Bliki: DesignStaminaHypothesis". `https://martinfowler.com/bliki/DesignStaminaHypothesis.html`.

23. Pramod Sadalage and Martin Fowler. "Evolutionary Database Design". `https://martinfowler.com/articles/evodb.html`.

24. Martin Fowler. "Bliki: FunctionAsObject". `https://martinfowler.com/bliki/FunctionAsObject.html`.

25. Martin Fowler. "Form Template Method". `https://refactoring.com/catalog/formTemplateMethod.html`.

26. Martin Fowler. "Bliki: ListAndHash". `https://martinfowler.com/bliki/ListAndHash.html`.

27. Martin Fowler. "The New Methodology". `https://martinfowler.com/articles/newMethodology.html`.

28. Martin Fowler. "Bliki: OverloadedGetterSetter". https://martinfowler.com/bliki/OverloadedGetterSetter.html.

29. Danilo Sato. "Bliki: ParallelChange". https://martinfowler.com/bliki/ParallelChange.html.

30. Martin Fowler. "Range". https://martinfowler.com/eaaDev/Range.html.

31. Martin Fowler. "Refactoring Code to Load a Document". https://martinfowler.com/articles/refactoring-document-load.html.

32. Martin Fowler. "Refactoring with Loops and Collection Pipelines". https://martinfowler.com/articles/refactoring-pipelines.html.

33. Martin Fowler. "Repository". https://martinfowler.com/eaaCatalog/repository.html.

34. Martin Fowler. "Bliki: SelfTestingCode". https://martinfowler.com/bliki/SelfTestingCode.html.

35. Martin Fowler. "Bliki: TestCoverage". https://martinfowler.com/bliki/TestCoverage.html.

36. Martin Fowler. "Bliki: TestDrivenDevelopment". https://martinfowler.com/bliki/TestDrivenDevelopment.html.

37. Martin Fowler. "Bliki: TypeInstanceHomonym". https://martinfowler.com/bliki/TypeInstanceHomonym.html.

38. Martin Fowler. "Bliki: UniformAccessPrinciple". https://martinfowler.com/bliki/UniformAccessPrinciple.html.

39. Martin Fowler. "Bliki: ValueObject". https://martinfowler.com/bliki/ValueObject.html.

40. Martin Fowler. "Bliki: ExtremeProgramming". https://martinfowler.com/bliki/ExtremeProgramming.html.

41. Martin Fowler. "Bliki: Xunit". https://martinfowler.com/bliki/Xunit.html.

42. Martin Fowler. "Bliki: Yagni". https://martinfowler.com/bliki/Yagni.html.

43. https://mochajs.org.

44. William F. Opdyke. "Refactoring Object-Oriented Frameworks". Doctoral Dissertation. University of Illinois at Urbana-Champaign, 1992. http://www.laputan.org/pub/papers/opdyke-thesis.pdf.

45. D. L. Parnas. "On the Criteria to Be Used in Decomposing Systems into Modules". In: *Communications of the ACM, Volume 15 Issue 12, pp. 1053–1058.* Dec. 1972.

46. https://refactoring.com.

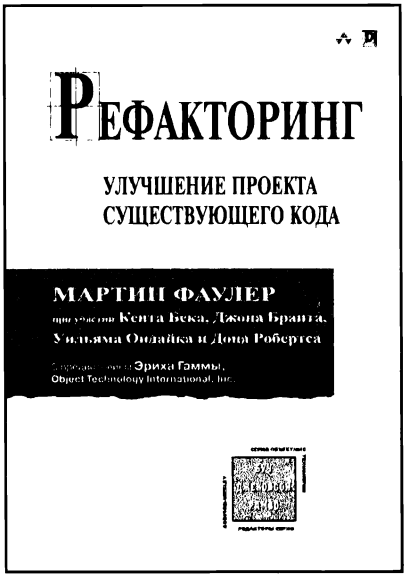
47. William C. Wake. *Refactoring Workbook.* Addison-Wesley, 2003. ISBN 0321109295.

48. Bill Wake. "The Swap Statement Refactoring". https://www.industriallogic.com/blog/swap-statement-refactoring/.

Предметный указатель

РЕФАКТОРИНГ
Улучшение проекта существующего кода

Мартин Фаулер



www.williamspublishing.com

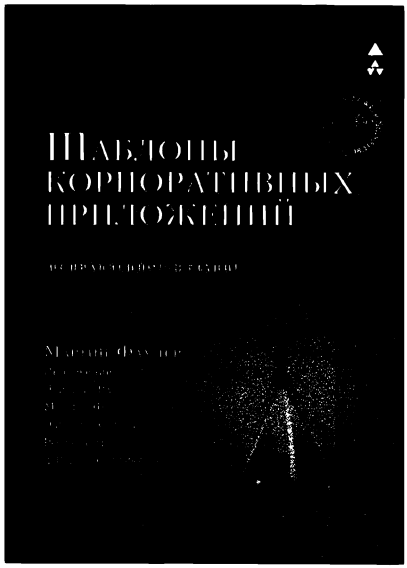
Мартин Фаулер и другие высококлассные специалисты в области объектно-ориентированного программирования, принявшие участие в написании этой книги, изложили принципы и наиболее эффективные методики выполнения различных рефакторингов и показали, когда и как следует тщательно изучать код с целью его улучшения. В книге подробно описано более 70 методов рефакторинга, причем приведено не только их теоретическое описание, но и практические примеры на языке программирования Java.

Данная классическая книга достойна того, чтобы занять свое место на книжной полке каждого серьезного программиста — вне зависимости от используемого языка программирования.

ISBN 978-5-9909445-1-0 **в продаже**

ШАБЛОНЫ КОРПОРАТИВНЫХ ПРИЛОЖЕНИЙ

Мартин Фаулер



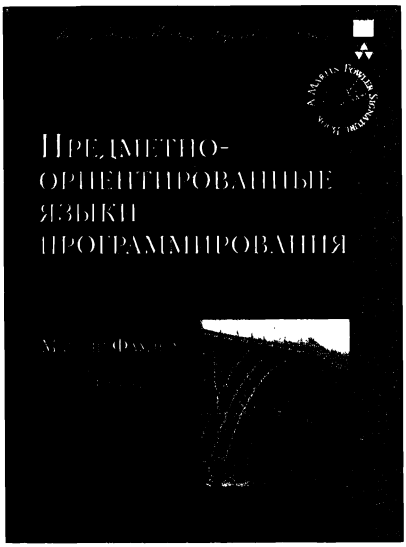
www.williamspublishing.com

Книга дает ответы на трудные вопросы, с которыми приходится сталкиваться всем разработчикам корпоративных систем. Автор, известный специалист в области объектно-ориентированного программирования, заметил, что с развитием технологий базовые принципы проектирования и решения общих проблем остаются неизменными, и выделил более 40 наиболее употребительных подходов, оформив их в виде типовых решений. Результат перед вами — незаменимое руководство по архитектуре программных систем для любой корпоративной платформы. Это своеобразное учебное пособие поможет вам не только усвоить информацию, но и передать полученные знания окружающим значительно быстрее и эффективнее, чем это удавалось автору. Книга предназначена для программистов, проектировщиков и архитекторов, которые занимаются созданием корпоративных приложений и стремятся повысить качество принимаемых стратегических решений.

ISBN 978-617-7812-08-0 **в продаже**

ПРЕДМЕТНО-ОРИЕНТИРОВАННЫЕ ЯЗЫКИ ПРОГРАММИРОВАНИЯ

Мартин Фаулер



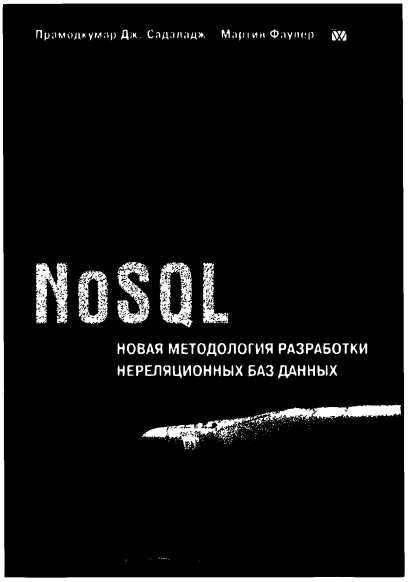
www.williamspublishing.com

В этой книге известный эксперт в области разработки программного обеспечения Мартин Фаулер предоставляет читателям всю информацию, необходимую для того, чтобы принять решение об использовании предметно-ориентированных языков в своих разработках и при положительном решении — эффективно применять методы создания таких языков. Каждая из методик сопровождается не только детальным пояснением ее работы, но и советами, когда именно ее стоит применять, а когда лучше прибегнуть к иным методам. Кроме того, здесь представлены примеры практического применения этих методик.

ISBN 978-5-8459-1738-6 **в продаже**

NoSQL: новая методология разработки нереляционных баз данных

Прамодкумар Дж. Садаладж
Мартин Фаулер



Необходимость обрабатывать все большие объемы данных является одним из факторов, стимулирующих появление альтернатив реляционным базам данных, использующим язык SQL. Одной из таких альтернатив является технология NoSQL.
В книге Фаулера рассматриваются основные концепции NoSQL, включая неструктурированные модели данных, агрегаты, новые модели распределения, теорему CAP и отображение-свертку, а также исследованы архитектурные и проектные вопросы, связанные с реализацией баз данных NoSQL.

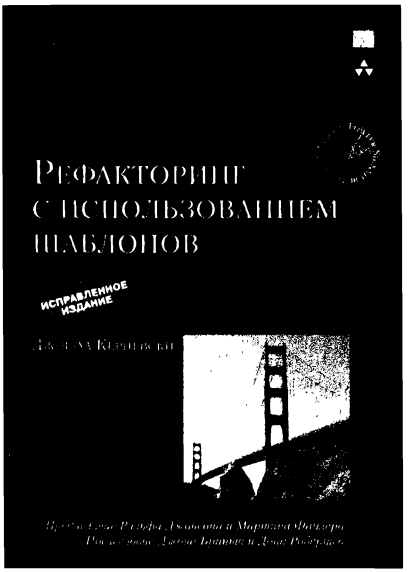
Книга предназначена для разработчиков баз данных, корпоративных приложений, а также для студентов.

www.williamspublishing.com

ISBN 978-5-8459-1920-5 в продаже

РЕФАКТОРИНГ С ИСПОЛЬЗОВАНИЕМ ШАБЛОНОВ

Джошуа Кериевски



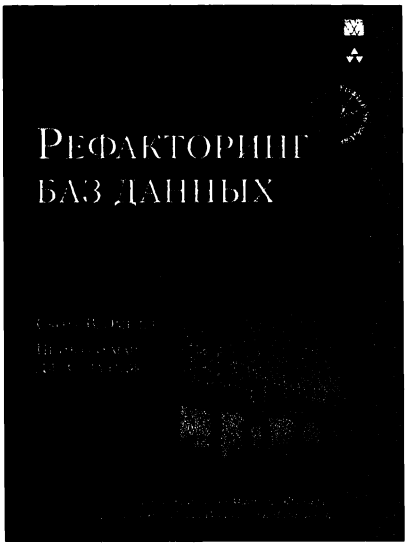
www.williamspublishing.com

Данная книга представляет собой результат многолетнего опыта профессионального программиста по применению шаблонов проектирования. Авторский подход к проектированию состоит в том, что следует избегать как недостаточного, так и избыточного проектирования, постоянно анализируя готовый работоспособный код и реорганизуя его только в том случае, когда это приведет к повышению его эффективности, упрощению его понимания и сопровождения. Автор на основании как собственного, так и чужого опыта детально рассматривает различные признаки кода, требующего рефакторинга, описывает, какой именно рефакторинг наилучшим образом подходит для той или иной ситуации, и описывает его механику, подробно разбирая ее на конкретных примерах из реальных задач. Книга может рассматриваться и как учебник по рефакторингу для программиста среднего уровня, и как справочное пособие для профессионала.

ISBN 978-5-8459-1087-5 **в продаже**

РЕФАКТОРИНГ БАЗ ДАННЫХ
ЭВОЛЮЦИОННОЕ ПРОЕКТИРОВАНИЕ

Скотт В. Эмблер,
Прамодкумар Дж. Садаладж



Книга посвящена описанию процедур проектирования базы данных с точки зрения архитектора объектно-ориентированного программного обеспечения, поэтому представляет интерес и для разработчиков прикладного кода, и для специалистов в области реляционных баз данных. Значительное место уделено описанию того, как действовать в тех практических ситуациях, когда база данных уже существует, но плохо спроектирована, или когда реализация первоначального проекта базы данных не позволила получить качественную модель. Прежде всего книгу можно использовать в качестве технического руководства для разработчиков, непосредственно занятых на производстве. С другой стороны, она представляет собой теоретическую работу, стимулирующую дальнейшие исследования в направлении объединения объектно-ориентированного и реляционного подходов.

www.williamspublishing.com

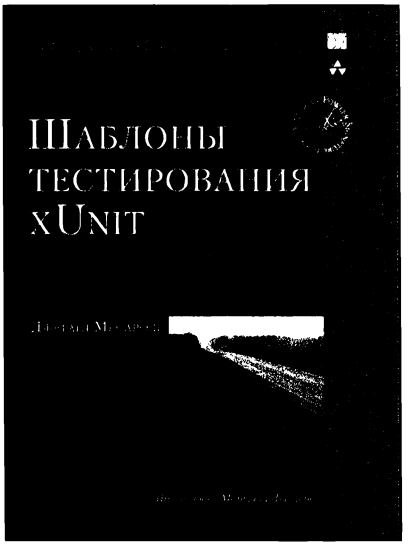
ISBN 978-5-8459-1157-5

в продаже

ШАБЛОНЫ ТЕСТИРОВАНИЯ XUNIT

РЕФАКТОРИНГ КОДА ТЕСТОВ

Джерард Месарош



www.williamspublishing.com

В данной книге показано, как применять принципы разработки программного обеспечения, в частности шаблоны проектирования, инкапсуляцию, исключение повторений и описательные имена, к написанию кода тестов. Книга состоит из трех частей.
В первой части приводятся теоретические основы методов разработки тестов, описываются концепции шаблонов и "запахов" тестов (признаков существующей проблемы). Во второй и третьей частях книги приводится каталог шаблонов проектирования тестов, "запахов" и других средств обеспечения большей прозрачности кода тестов. Кроме этого, в третьей части книги сделана попытка обобщить и привести к единому знаменателю терминологию тестовых двойников и подставных объектов, а также рассмотрены некоторые принципы их применения при проектировании как тестов, так и самого программного обеспечения.
Книга ориентирована на разработчиков программного обеспечения, практикующих гибкие процессы разработки.

ISBN 978-5-8459-1448-4

в продаже

ЭФФЕКТИВНАЯ РАБОТА С УНАСЛЕДОВАННЫМ КОДОМ

Майкл К. Физерс

www.williamspublishing.com

Эта книга посвящена практическим вопросам эффективной работы с унаследованным кодом. В ней освещаются механизмы внесения изменений в унаследованный код, способы переноса его фрагментов в среду тестирования, особенности написания тестов для безопасного изменения и реорганизации кода, приемы точного определения мест для подобных изменений, а также подходы к обращению с унаследованным процедурным кодом. Кроме того, в книге представлены способы разрыва зависимостей для работы с обособленными фрагментами кода и безопасного внесения в них изменений.
Книга адресована тем, кто имеет опыт разработки прикладного программного обеспечения и его сопровождения.

ISBN 978-5-8459-1530-6 **в продаже**

Запахи в коде

Запах	Применяемые рефакторинги
Таинственное имя (с. 112)	*Изменение объявления функции* (с. 170), *Переименование переменной* (с. 170), *Переименование поля* (с. 289)
Дублируемый код (с. 112)	*Извлечение функции* (с. 152), *Перемещение инструкций* (с. 269), *Подъем метода* (с. 394)
Длинная функция (с. 113)	*Извлечение функции* (с. 152), *Замена временной переменной запросом* (с. 225), *Введение объекта параметра* (с. 186), *Сохранение всего объекта* (с. 364), *Замена функции командой* (с. 381), *Декомпозиция условной инструкции* (с. 306), *Замена условной инструкции полиморфизмом* (с. 317), *Разделение цикла* (с. 274)
Длинный список параметров (с. 114)	*Замена параметра запросом* (с. 369), *Сохранение всего объекта* (с. 364), *Введение объекта параметра* (с. 186), *Удаление аргумента-флага* (с. 359), *Объединение функций в класс* (с. 190)
Глобальные данные (с. 115)	*Инкапсуляция переменной* (с. 178)
Изменяемые данные (с. 116)	*Инкапсуляция переменной* (с. 178), *Расщепление переменной* (с. 285), *Перемещение инструкций* (с. 269), *Извлечение функции* (с. 152), *Отделение запроса от модификатора* (с. 352), *Удаление метода установки значения* (с. 376), *Замена вычисленной переменной запросом* (с. 293), *Объединение функций в класс* (с. 190), *Объединение функций в преобразование* (с. 195), *Замена ссылки значением* (с. 296)
Расходящиеся изменения (с. 117)	*Разделение этапа* (с. 201), *Перенос функции* (с. 244), *Извлечение функции* (с. 152), *Извлечение класса* (с. 229)
Стрельба дробью (с. 118)	*Перенос функции* (с. 244), *Перенос поля* (с. 253), *Объединение функций в класс* (с. 190), *Объединение функций в преобразование* (с. 195), *Разделение этапа* (с. 201), *Встраивание функции* (с. 161), *Встраивание класса* (с. 232)
Завистливые функции (с. 118)	*Перенос функции* (с. 244), *Извлечение функции* (с. 152)
Группы данных (с. 119)	*Извлечение класса* (с. 229), *Введение объекта параметра* (с. 186), *Сохранение всего объекта* (с. 364)

Запах	Применяемые рефакторинги
Одержимость примитивами (с. 120)	*Замена примитива объектом* (с. 221), *Замена кода типа подклассами* (с. 405), *Замена условной инструкции полиморфизмом* (с. 317), *Извлечение класса* (с. 229), *Введение объекта параметра* (с. 186)
Повторяющиеся switch (с. 121)	*Замена условной инструкции полиморфизмом* (с. 317)
Циклы (с. 121)	*Замена цикла конвейером* (с. 278)
Ленивый элемент (с. 122)	*Встраивание функции* (с. 161), *Встраивание класса* (с. 232), *Свертывание иерархии* (с. 423)
Теоретическая общность (с. 122)	*Свертывание иерархии* (с. 423), *Встраивание функции* (с. 161), *Встраивание класса* (с. 232), *Изменение объявления функции* (с. 170), *Удаление неработающего кода* (с. 283)
Временное поле (с. 123)	*Извлечение класса* (с. 229), *Перенос функции* (с. 244), *Введение частного случая* (с. 334)
Цепочки сообщений (с. 123)	*Сокрытие делегата* (с.235), *Извлечение функции* (с. 152), *Перенос функции* (с. 244)
Посредник (с. 124)	*Удаление посредника* (с. 238), *Встраивание функции* (с. 161), *Замена суперкласса делегатом* (с. 443), *Замена подкласса делегатом* (с. 424)
Внутренний обмен (с. 124)	*Перенос функции* (с. 244), *Перенос поля* (с. 253), *Сокрытие делегата* (с. 235), *Замена подкласса делегатом* (с. 424), *Замена суперкласса делегатом* (с. 443)
Большой класс (с. 125)	*Извлечение класса* (с. 229), *Извлечение суперкласса* (с. 418), *Замена кода типа подклассами* (с. 405)
Альтернативные классы с разными интерфейсами (с. 125)	*Изменение объявления функции* (с. 170), *Перенос функции* (с. 244), *Извлечение суперкласса* (с. 418)
Классы данных (с. 126)	*Инкапсуляция записи* (с. 208), *Удаление метода установки значения* (с. 376), *Перенос функции* (с. 244), *Извлечение функции* (с. 152), *Разделение этапа* (с. 201)
Отказ от наследства (с. 126)	*Опускание метода* (с. 403), *Опускание поля* (с. 404), *Замена подкласса делегатом* (с. 424), *Замена суперкласса делегатом* (с. 443)
Комментарии (с. 127)	*Извлечение функции* (с. 152), *Изменение объявления функции* (с. 170), *Введение утверждения* (с. 346)

Программная инженерия

Рефакторинг кода на JavaScript

Полностью пересмотренное и обновленное издание, включает новые рефакторинги и примеры кода

Более двадцати лет опытные программисты во всем мире использовали книгу Мартина Фаулера *Рефакторинг*, когда им нужно было улучшить проект существующего кода, повысить удобство сопровождения программного обеспечения или облегчить понимание существующего кода. Это — такое долгожданное — новое издание было полностью обновлено, чтобы отразить важные изменения в области программирования. Второе издание книги содержит обновленный каталог рефакторингов и включает примеры кода на JavaScript, а также новые функциональные примеры, демонстрирующие рефакторинг без классов.

Как и в первом издании, здесь объясняется, что такое рефакторинг, почему вы должны прибегать к нему, как распознать код, который нуждается в рефакторинге, и как успешно провести его независимо от того, какой язык программирования вы используете.

ОСНОВНЫЕ ТЕМЫ КНИГИ

- Понимание процесса и общих принципов рефакторинга
- Быстрое применение полезных рефакторингов для облегчения понимания и изменения программ
- Распознавание запаха в коде, который сигнализирует о возможном применении рефакторинга
- Каталог рефакторингов с объяснениями, мотивацией, техникой применения и простыми примерами
- Создание надежных тестов для рефакторингов
- Компромиссы и препятствия на пути рефакторинга

МАРТИН ФАУЛЕР — главный научный сотрудник компании ThoughtWorks. Он называет себя “автором, докладчиком, консультантом и просто человеком, болтающим о разработке программного обеспечения”. Фаулер занимается разработкой программного обеспечения для предприятий, изучая, что делает проект хорошим и какие методы необходимо применять для его создания.

Диалектика
www.williamspublishing.com

Pearson
Addison-Wesley

ISBN 978-5-907144-59-0