1 Рефакторинг рефакторинга

В этой главе

- ✓ Разбор элементов рефакторинга.
- ✓ Внедрение рефакторинга в вашу повседневную работу.
- ✓ Важность безопасности при рефакторинге.
- ✓ Знакомство с общим примером для всей части I книги.

Ни для кого не секрет, что высокое качество кода подразумевает его более эффективное обслуживание, меньше ошибок и хорошее настроение разработчиков. Наиболее распространенный способ добиться от кода высокого качества — это рефакторинг. Тем не менее тот метод, которым обычно обучают рефакторингу — с *запахами кода* и *модульным тестированием*, — воздвигает для начинающих очень высокий порог вхождения. Я же считаю, что после небольшой практики любой сможет безопасно применить простые шаблоны рефакторинга.

В разработке ПО мы распределяем задачи по схеме, изображенной на рис. 1.1. Таким образом обозначаются необходимые для их решения предпосылки: навыки, инструменты, культура или их комбинация. Рефакторинг — это сложная технология, которая располагается в середине. Для его выполнения требуются все упомянутые компоненты.

- *Навыки* — нужны, чтобы выявить плохой код, требующий рефакторинга. Опытные программисты могут сделать это на основе собственных знаний

через запахи кода. Однако сами запахи зачастую не имеют четких границ и определений (требуют опытного анализа), их интерпретация субъективна, в связи с чем их сложно заучить. Поэтому для начинающего разработчика обнаружение запахов кода больше напоминает включение шестого чувства, чем навык.

- *Культура* — стремление обращаться к рефакторингу должно быть заложено в культуру и рабочую повседневность. Во многих случаях такая культура реализуется посредством известного цикла *«красный — зеленый — рефакторинг»*, используемого в разработке через тестирование. Однако разработка через тестирование — это, как мне кажется, еще более сложное искусство. К тому же цикл «красный — зеленый — рефакторинг» не предоставляет удобного способа рефакторить старые (legacy) базы кода.
- *Инструменты* — нам требуется средство, которое бы позволило убедиться в безопасности выполняемых действий. Самый распространенный способ сделать это — автоматизированное тестирование. Но, как я уже сказал, выработка хорошего навыка в тестировании сама по себе представляет сложность.

Рис. 1.1. Навыки, культура и инструменты

В последующих разделах мы углубимся в каждую из этих областей и обрисуем, как можно начать путешествие в мир рефакторинга с упрощенных основ, без тестирования и абстрактных запахов кода. Изучение рефакторинга подобным образом способно быстро продвинуть качество кода начинающих разработчиков,

студентов и программистов-любителей на новый уровень. Технические руководители могут использовать методы из этой книги для обучения рефакторингу своих подчиненных, которые пока не привыкли применять его в производственном процессе.

1.1. ЧТО ТАКОЕ РЕФАКТОРИНГ

На этот вопрос я развернуто отвечу в следующей главе, хотя получить некоторое представление заранее тоже будет полезным. В простейшей форме *рефакторинг* означает «внесение в код изменений без изменения его действий». Для наглядности начнем с примера. В нем мы заменяем выражение локальной переменной.

Листинг 1.1. До

```
return pow(base, exp / 2) * pow(base, exp / 2);
```

Листинг 1.2. После

```
let result = pow(base, exp / 2);
return result * result;
```

Для рефакторинга может быть много причин:

- ускорение работы кода (как в приведенном примере);
- уменьшение объема кода;
- придание коду универсальности или возможности повторного использования;
- облегчение восприятия и обслуживания кода.

Последняя причина является основной и очень важной, поэтому ее можно сформулировать как «создание хорошего кода».

ОПРЕДЕЛЕНИЕ

Хороший код — это такой код, который не просто корректно выполняет свои задачи, но также удобен для чтения и прост в обслуживании.

Поскольку рефакторинг не должен изменять функционал кода, то в течение книги мы сосредоточимся на его читаемости и обслуживаемости. Эти причины для рефакторинга будут разобраны более подробно в главе 2. В книге мы будем рассматривать только такой рефакторинг, который приводит к хорошему коду. В связи с этим для него будет актуальным следующее определение.

ОПРЕДЕЛЕНИЕ

Рефакторинг — изменение кода с целью повышения его читаемости и обслуживаемости без изменения функциональности.

Должен также сказать, что тип рассматриваемого нами рефакторинга во многом опирается на работу с языком объектно-ориентированного программирования.

Многие представляют себе программирование как написание кода. Однако обычно программисты больше времени проводят за чтением кода и его разбором, чем за написанием. А происходит это потому, что слишком сложна сфера, в которой они работают, и внесение изменений без досконального понимания кода может привести к катастрофическим сбоям.

Итак, первый аргумент для рефакторинга чисто экономический: время программиста стоит дорого, поэтому если мы сделаем нашу базу кода более читаемой, то выиграем время для реализации следующим программистом новых возможностей. Второй аргумент состоит в том, что повышение обслуживаемости кода означает меньшее число ошибок, которые к тому же проще исправить. Третий аргумент гласит, что с хорошей базой кода просто интереснее работать. Когда мы читаем код, то моделируем в своем сознании его действия. Чем больше информации нам приходится удерживать в голове одновременно, тем более утомительным становится этот процесс. Именно поэтому отладка может оказаться ужасной и возникает соблазн заново переделать все полностью.

1.2. НАВЫКИ: ЧТО ТРЕБУЕТ РЕФАКТОРИНГА

Распознать, где именно необходимо применить рефакторинг, — первая трудность. Обычно рефакторингу обучают в сочетании с так называемыми *запахами кода*. Эти запахи или запашки представляют описания того, как может проявлять себя плохое качество кода. Ориентирование по запахам является мощной техникой, но в то же время сами запахи довольно абстрактны, и начинать рефакторинг с этого сложно. Здесь требуется чутье, которое вырабатывается только с опытом.

Книга, которую вы держите в руках, использует другой подход, представляя легко узнаваемые и простые в применении правила для определения требующих рефакторинга фрагментов. Эти правила не только легко использовать, но и просто запомнить. При этом иногда они могут оказаться излишне строгими и потребуют от вас исправить код, который не пахнет. В редких же случаях, наоборот, мы можем последовать этим правилам и после их применения все равно получить код с запашком.

Как показано на рис. 1.2, соответствие между запахами кода и правилами рефакторинга неидеально. Мои правила не являются необходимым и достаточным условием хорошего кода. Они представляют собой начало пути к выработке профессионального понимания того, какой именно код можно

посчитать хорошим. Предлагаю рассмотреть пример различия между оценками кода по запаху и по правилам этой книги.

Рис. 1.2. Правила и запахи кода

1.2.1. Пример запаха кода

К известным запашкам относится, например, такой: функция должна выполнять *одно* действие. Это отличный ориентир, но бывает сложно понять, что именно является этим *одним*. Еще раз взглянем на приведенный выше код: он пахнет? Возможно, так как он делит, вычисляет экспоненту, а затем умножает. Означает ли это, что он совершает три действия? С другой стороны, он просто возвращает одно число и не меняет никакого состояния. Так, может, он все же совершает одно действие?

```
let result = pow(base, exp / 2);
return result * result;
```

1.2.2. Пример правила

Сравним предыдущий запашок со следующим правилом (подробно оно будет разобрано в главе 3): метод никогда не должен содержать больше *пяти строк кода*. Это можно определить на глаз, не задавая лишних вопросов. Правило простое, лаконичное и легко запоминается, особенно ввиду того, что так называется эта книга.

Запомните, что правила из этой книги — своеобразные подстраховки. Как уже говорилось, они не могут гарантировать хороший код во всех ситуациях. В некоторых случаях следование им может оказаться ошибочным. Однако эти

правила оказываются полезными, когда вы не знаете, с чего начать, и обычно инициируют красивый рефакторинг кода.

Заметьте, что все названия правил выражены в абсолютной категоричной форме и в них используются слова вроде *«никогда»*: это упрощает запоминание. Однако в их подробных описаниях зачастую указываются исключения — ситуации, когда эти правила *не* рекомендуется применять. В описаниях также приводятся назначения правил. В начале изучения рефакторинга нужно использовать только абсолютную форму правил. После того как мы их усвоим, можно будет перейти к изучению исключений и лишь затем опираться уже на их базовое назначение — вот тогда мы станем настоящими гуру в написании кода.

1.3. КУЛЬТУРА: КОГДА ПРОВОДИТЬ РЕФАКТОРИНГ

Рефакторинг подобен принятию душа.

Кент Бек

Рефакторинг наиболее эффективен и наиболее экономичен при регулярном проведении. Поэтому, если у вас есть такая возможность, то советую внедрить эту практику в повседневную работу. В большинстве литературных источников предлагается использовать поток «красный — зеленый — рефакторинг». Но, как уже говорилось, этот подход привязывает рефакторинг к разработке через тестирование, а в данной книге нам нужно эти концепции разделить и сосредоточиться именно на рефакторинге, исключая тестирование из подробного рассмотрения. Следовательно, я советую использовать для решения любой задачи программирования более обобщенный рабочий поток, схематично показанный на рис. 1.3.

1. *Исследование*. Нередко в самом начале мы еще не до конца понимаем, что от нас требуется. Бывает, что заказчик сам не полностью осознает, чего именно хочет, или его требования прописаны неоднозначно. Еще бывает, что сразу и не разберешь, выполнимая ли вообще перед тобой задача. Поэтому всегда начинайте с экспериментирования. Реализуйте примерный образец, и у вас уже будет на чем строить диалог с заказчиком, уточняя его требования и ваши возможности.
2. *Специфицирование*. После того как вы утвердили задачу, оформите ее четкие критерии. В идеале это делается в форме автоматизированного тестирования.
3. *Реализация*. Реализуйте код.
4. *Тестирование*. Убедитесь, что код соответствует спецификации из шага 2.

5. *Рефакторинг.* Перед отправкой кода убедитесь, что в дальнейшем другим разработчикам будет удобно с ним работать (к тому же этим разработчиком можете снова стать вы).
6. *Доставка.* Для доставки кода существует множество способов; наиболее распространенный — это создание пула реквеста или отправка его в определенную ветку. Самое главное, чтобы ваш код попал к пользователям. Иначе какой вообще в нем смысл?

Рис. 1.3. Поток разработки

Поскольку мы проводим рефакторинг *на основе правил*, то рабочий поток весьма прост и к нему легко будет приступить. На рис. 1.4 подробно развернут шаг 5: *рефакторинг*.

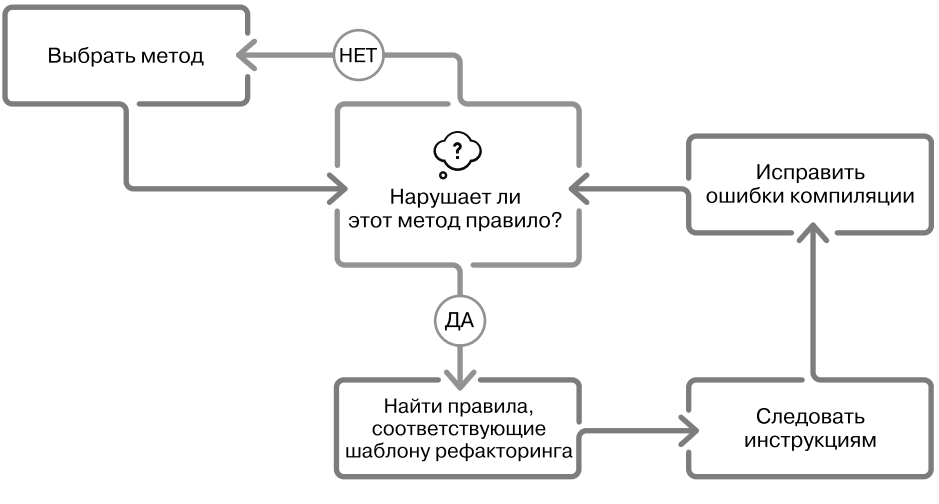

Рис. 1.4. Подробное представление этапа рефакторинга

Я разработал правила так, чтобы они быстро запоминались и можно было легко обнаружить, где требуется их применение и как его провести. Это значит, что без дополнительных ориентиров несложно идентифицировать метод, нарушающий правило. При этом с каждым правилом связано несколько шаблонов рефакторинга, что помогает понять, как именно применить правило для исправления обнаруженной проблемы. Шаблоны рефакторинга содержат отчетливые пошаговые инструкции, чтобы вы по случайности ничего не сломали. Многие из приводимых в этой книге шаблонов намеренно задействуют ошибки компиляции, чтобы вы гарантированно не внесли каких-либо нарушений. После того как мы немного попрактикуем, и правила, и шаблоны рефакторинга станут для вас обыденным делом.

1.3.1. Рефакторинг старых унаследованных систем

Предположим, что мы начинаем работать с обширной legacy-системой. И в этом случае есть грамотный способ внедрить в повседневную работу рефакторинг без необходимости прерывать весь рабочий поток, чтобы сначала полностью отрефакторить базу кода. Достаточно следовать такой чудесной рекомендации.

> Сначала сделайте изменение простым, а затем уже внесите это простое изменение.
>
> *Кент Бек*

Итак, если мы собираемся реализовать нечто новое, то должны начать с рефакторинга, это упрощает добавление дополнительного кода. Просматривается аналогия с подготовкой ингредиентов перед замешиванием теста для выпечки.

1.3.2. Когда рефакторинг делать не нужно

Чаще всего рефакторинг — это здорово, но есть у него и обратные стороны. Он может занимать много времени, особенно если проводить его нерегулярно. Ну а время программиста, как мы знаем, обходится недешево.

Есть три типа баз кода, в которых рефакторинг, скорее всего, себя не оправдает.

- Код, который вы планируете написать, выполнить один раз и удалить. В сообществе экстремального программирования (Extreme Programming) это называется *«спайк» (spike)*.
- Код, который находится на обслуживании в преддверии завершения его эксплуатации.

- Код со строгими требованиями к производительности, например встраиваемая система или передовой физический движок в игре.

Во всех других случаях вложенные в рефакторинг силы и время определенно себя оправдают.

1.4. ИНСТРУМЕНТЫ: КАК ПРОВОДИТЬ РЕФАКТОРИНГ (БЕЗОПАСНО)

Мне, как и всем, нравятся автоматизированные тесты. Однако выработка хорошего навыка тестирования сама по себе представляется сложной. Поэтому, если вы уже умеете выполнять такие тесты, то можете смело использовать эту технику при прочтении книги. Если же нет, то ничего страшного.

Автоматизированные тесты в разработке ПО сродни тормозам у автомобилей. Мы оборудуем авто тормозной системой не потому, что хотим ехать медленно, а для того, чтобы быстрая езда была безопасной. То же верно и для программного обеспечения: автоматизированные тесты позволяют нам двигаться быстро и при этом безопасно. В этой книге мы изучаем абсолютно новый навык, значит, спешить нежелательно.

Вместо этого я предлагаю опереться на другие инструменты, например:

- детальные, поэтапные, структурированные шаблоны рефакторинга, подобные четким рецептам;
- контроль версий;
- компилятор.

На мой взгляд, если шаблоны рефакторинга спроектировать тщательно и выполнять мелкими шажками, то можно провести рефакторинг, ничего не нарушив. Это особенно актуально в случаях, когда наша IDE (интегрированная среда программирования, Integrated Development Environment) способна выполнять рефакторинг за нас.

Чтобы компенсировать отсутствие тестирования, в этой книге мы будем вылавливать многие распространенные ошибки с помощью компилятора и типов. Но я все равно рекомендую вам периодически открывать приложение, над которым вы работаете, и проверять его функциональность. Когда мы это проверим или компилятор не сообщит об ошибках, можно делать для себя коммит. Таким образом, если в дальнейшем работоспособность приложения нарушится и мы не сможем это с ходу исправить, можно будет откатиться к последнему коммиту.

Работая с реальными системами без автоматизированных тестов, мы все равно можем выполнять рефакторинг, но при этом наша уверенность должна будет на что-то опираться. В качестве такой опоры можно использовать для проведения рефакторинга IDE, выполнять ручные тесты, продвигаться очень мелкими шажками и т. п. Однако если учесть, что для этого необходимо затратить дополнительное время, то наверняка окажется целесообразнее использовать именно автоматизированное тестирование.

1.5. ИНСТРУМЕНТЫ, НЕОБХОДИМЫЕ ДЛЯ НАЧАЛА

Как я уже сказал, способы рефакторинга из этой книги требуют использования объектно-ориентированного языка программирования (ООП-языка). Это первое, что вам нужно, чтобы читать и понимать ее.

И написание кода, и рефакторинг представляют собой деятельность, которую мы осуществляем непосредственно на компьютере, своими руками. Следовательно, и осваивать эти навыки лучше всего, работая на компьютере, следуя приводимым примерам и экспериментируя. Для этого вам потребуются инструменты, описанные далее. Инструкции по их установке можно найти в приложении в конце книги.

1.5.1. Язык программирования: TypeScript

Все примеры кода книги написаны на TypeScript. Этот язык я выбрал по ряду причин. Самое главное — это то, что он выглядит и ощущается родственным многим распространенным языкам — Java, C#, C++ и JavaScript. Это позволит читателям, знакомым с любым из них, читать и понимать листинги без каких-либо проблем. Кроме того, TypeScript предоставляет способ перейти от полностью «не объектно-ориентированного» кода (то есть кода без классов) к его объектно-ориентированной альтернативе.

ПРИМЕЧАНИЕ

С целью оптимизации использования пространства в печатной версии книги в ней используется стиль программирования, в котором исключаются переводы строки, но сохраняется читаемость. Я не призываю вас следовать такому же стилю, если только вы по случайности тоже не пишете книгу с большим количеством TS-кода. По этой же причине отступы и скобки иногда имеют различное форматирование в книге и в коде проекта.

На случай, если TypeScript вам не знаком, по мере появления каких-то его нюансов я буду пояснять их вот в таких вставках.

В TYPESCRIPT...

Для проверки равенства используется оператор тождественности (===), потому что его действие больше походит на ожидаемый эффект равенства, чем действие нестрогого равенства (==). Рассмотрим пример:

`0 == ""` верно.

`0 === ""` ложно.

Несмотря на то что примеры приводятся на TypeScript, все шаблоны рефакторинга и правила являются общими и применимы к любому ООП-языку. В редких случаях TypeScript как таковой проявляет собственные нетипичные для других языков особенности, помогает или мешает нам. Эти случаи отмечены явно, и я объясняю, что делать в подобных ситуациях в других популярных языках.

1.5.2. Редактор: Visual Studio Code

Я не предполагаю, что вы будете использовать определенную программу-редактор. Тем не менее если у вас нет личных предпочтений, то советую выбрать Visual Studio Code. Он отлично работает с TypeScript, а также поддерживает работу `tsc -w` в фоновом режиме, автоматически выполняя компиляцию, чтобы мы не забыли о ней.

ПРИМЕЧАНИЕ

Visual Studio Code совершенно не то же самое, что *Visual Studio*.

1.5.3. Контроль версий: Git

Несмотря на то что для следования материалу книги использовать контроль версий не обязательно, я настоятельно его рекомендую, поскольку так будет намного проще отменить какие-либо действия, если вы запутаетесь в процессе.

СБРОС К ЭТАЛОННОМУ РЕШЕНИЮ

Вы можете в любой момент вернуть код к тому виду, в каком он был в начале основного раздела, с помощью, например, такой команды:

```
git reset --hard section-2.1
```

Предупреждение: при этом все внесенные вами изменения будут потеряны.

1.6. ОБЩИЙ ПРИМЕР: 2D-ГОЛОВОЛОМКА

В завершение опишу сам способ, которым буду обучать вас всем этим прекрасным правилам и крутым шаблонам рефакторинга. Книга построена вокруг одного общего примера: 2D-головоломки с перемещением блоков, она похожа на классическую игру Boulder Dash (рис. 1.5).

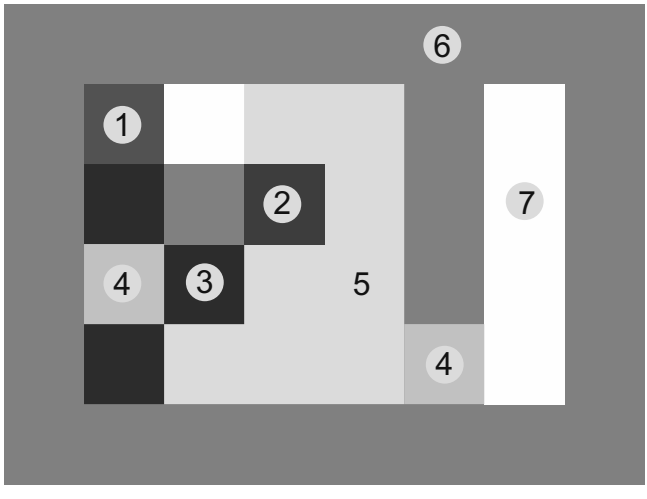

Рис. 1.5. Снимок экрана игры «из коробки»

Это означает, что на всю часть I книги у нас будет единая массивная база кода. Использование всего одного примера сэкономит время, потому что не придется в каждой главе знакомиться с новым.

Сам пример написан в реалистичном стиле, похожем на используемый в индустрии. Простым это упражнение назвать нельзя, если только вы уже не владеете навыками, описываемыми в данной книге. Код изначально подчиняется принципам *DRY* («Не повторяйся») и *KISS* («Не усложняй»). Но даже в таком виде он не приятнее сухого поцелуя[1].

Я выбрал компьютерную игру, потому что при ручном тестировании так будет проще заметить некорректное поведение кода на основе интуитивного понимания того, как должен происходить сам процесс игры. К тому же так тестирование окажется веселее по сравнению с просматриванием, например, логов финансовых систем.

Пользователь управляет клеткой игрока с помощью клавиш-стрелок. Цель игры — передвинуть ящик (на рис. 1.5 обозначен как 2) в нижний правый угол.

---

[1] Авторская игра слов, так как в английском dry — это «сухой», а kiss — это «поцелуй». — *Здесь и далее примеч. пер.*

Несмотря на то что печатная версия книги черно-белая, на деле элементы представлены разными цветами.

1. Красная клетка — это игрок.
2. Коричневая клетка — это ящик.
3. Синие — это камни.
4. Желтые представляют ключи или замки — с этим мы разберемся позже.
5. Зеленоватые клетки называются *флаксами*.
6. Серые — это стены.
7. Белые — это воздух (пустота).

Если ящик или камень ничем не поддерживаются, то они падают. Игрок может передвинуть один камень или ящик за один ход при условии, что этот предмет не огражден и в результате не упадет. Путь между ящиком и нижним правым углом изначально закрыт на замок, поэтому игроку нужно сначала найти ключ. Флакс можно «съесть» (удалить), наступив на него.

Теперь самое время скачать игру и поэкспериментировать.

1. Перейдите на компьютере в расположение, куда хотите сохранить игру, и откройте консоль:
   1) команда `git clone https://github.com/thedrlambda/five-lines` скачает ее исходный код;
   2) команда `tsc -w` будет компилировать TypeScript в JavaScript при каждом его изменении.
2. Откройте `index.html` в браузере.

   Здесь у вас есть возможность через код изменять уровни, так что смело экспериментируйте с созданием собственных карт путем обновления массива в переменной `map` (пример можете найти в приложении).
3. Откройте каталог в Visual Studio Code.
4. Выберите Terminal, затем New Terminal.
5. Выполните `tsc -w`.
6. Теперь TypeScript будет компилировать изменения в фоновом режиме, и терминал можно закрыть.
7. После внесения каждого изменения подождите секунду, пока TypeScript выполнит компиляцию, а затем обновите содержимое окна браузера.

Такую же процедуру вы будете использовать при написании кода в соответствии с примерами части I книги. Но прежде, чем к этому перейти, в следующем разделе мы заложим более детальную основу рефакторинга.

1.6.1. Практика ведет к совершенству: вторая база кода

Я свято верю в силу практики и поэтому создал еще один проект, для которого решения не привел. Этот проект вы можете использовать при перечитывании книги, для испытания собственных навыков или в качестве упражнения для студентов, если вы преподаватель. Он представляет собой 2D-игру в жанре экшена. В обеих базах кода (основного примера и дополнительного проекта) используются одинаковые стиль, структура, элементы. При этом их рефакторинг подразумевает выполнение одних и тех же шагов. Несмотря на то что вторая база кода более продвинутая, внимательно следуя правилам и шаблонам рефакторинга, вы должны добиться желаемого результата и в ней. Для скачивания второго проекта выполните все те же шаги, но со следующим URL: https://github.com/thedrlambda/bomb-guy.

1.7. ПРИМЕЧАНИЕ ПО РЕАЛЬНЫМ ПРОГРАММАМ

Важно еще раз повторить, что основной задачей этой книги является введение в работу рефакторинга. Эта задача не подразумевает указание конкретных правил, которые вы могли бы применять в продакшене в той или иной ситуации. Для их использования сначала нужно выучить названия правил и повторять их. Когда для вас это станет простым, изучите описания, отражающие исключения из правил. Наконец, используйте все это для выработки понимания лежащих в основе правил запахов кода. Предстоящий вам путь схематично показан на рис. 1.6.

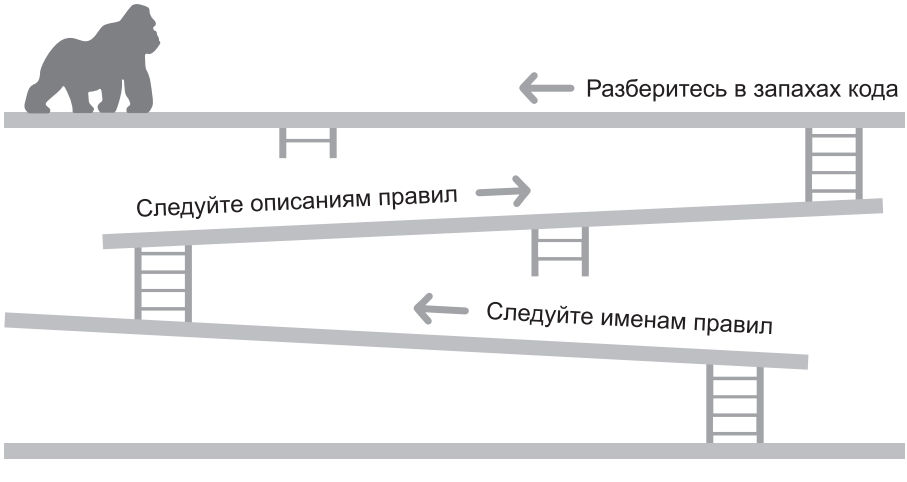

Рис. 1.6. Как использовать правила

Необходимость именно такого постижения смысла правил и неформальный подход к рефакторингу объясняет невозможность создания программы автоматического рефакторинга. (Хотя можно разработать плагин, который на основе правил будет выделять *предположительно проблематичные* области кода.) Цель же самих правил — выработать понимание. Если коротко: следуйте правилам, пока сами не начнете понимать, как лучше.

Имейте также в виду, что благодаря концентрации исключительно на изучении рефакторинга и наличию безопасной среды при чтении книги можно обойтись без автоматизированных тестов, но вряд ли это пройдет при работе в реальных системах. Здесь мы поступаем так, потому что изучать рефакторинг и автоматизированное тестирование гораздо эффективнее по отдельности.

РЕЗЮМЕ

- Проведение рефакторинга требует наличия трех предпосылок: *навыков* для определения фрагментов кода, требующих улучшения, *культуры* для понимания, когда нужно это делать, и *инструментов*, позволяющих понять, как именно нужно это делать.
- Условно для определения того, что именно нужно рефакторить, используются запахи кода. Начинающим программистам сложно разобраться в них, не имея опыта, потому что они слишком абстрактны. В этой книге приводятся конкретные правила, способные на время обучения заменить практику определения запахов кода. Эти правила имеют три уровня абстракции: конкретные названия, собственно описания, касающиеся смысловых нюансов в виде исключений, и, наконец, сама суть запахов.
- Я верю, что для снижения порога вхождения в автоматизированное тестирование и рефакторинг нужно изучать эти дисциплины по отдельности. Вместо автоматизированного тестирования мы задействуем компилятор, контроль версий и ручное тестирование.
- Рабочий поток рефакторинга обычно связан с разработкой через тестирование в цикле «красный — зеленый — рефакторинг». Но это опять же подразумевает зависимость от автоматизированных тестов. Вместо этого я предлагаю использовать рабочий поток из шести шагов (исследование, специфицирование, реализация, тестирование, рефакторинг, доставка) как для нового кода, так и для рефакторинга непосредственно перед изменением старого.

На протяжении части I книги мы будем изменять исходный код 2D-головоломки с использованием Visual Studio Code, TypeScript и Git.