25 Шаблоны разработки через тестирование

Прежде чем приступить к обсуждению эффективных методов тестирования, давайте попробуем ответить на несколько стратегических вопросов:

- Что такое тестирование?
- Когда мы выполняем тестирование?
- Какая логика нуждается в тестировании?
- Какие данные нуждаются в тестировании?

Тест

Каким образом следует тестировать программное обеспечение? При помощи автоматических тестов.

Тестировать означает *проверять*. Ни один программист не считает работу над некоторым фрагментом кода завершенной, не проверив его работоспособность (исключение составляют либо слишком самоуверенные, либо слишком небрежные программисты, но я надеюсь, что среди читателей данной книги таких нет). Однако, если вы тестируете свой код, это не означает, что у вас *есть* тесты. *Тест* — это процедура, которая позволяет либо подтвердить, либо опровергнуть работоспособность кода. Когда программист проверяет работоспособность разработанного им кода, он выполняет тестирование вручную: нажимает кнопки на клавиатуре и смотрит на результат работы программы, отображаемый на экране. В данном контексте тестирование состоит из двух этапов: запуск кода и проверка результатов его работы. *Автоматический тест* выполняется автоматически: вместо программиста запуском кода и проверкой результатов занимается компьютер, который отображает на экране результат выполнения теста: *код работоспособен* или *код неработоспособен*. В чем состоит принципиальное отличие автоматического теста от тестирования кода вручную?

На рис. 25.1 представлена диаграмма взаимовлияния между стрессом и тестированием (она напоминает диаграммы Герри Вейнберга (Gerry Weinberg) в его книге *Quality Software Management*). Стрелка между узлами диаграммы означает, что увеличение первого показателя влечет за собой увеличение второго показателя. Стрелка с кружком означает, что увеличение первого показателя влечет за собой уменьшение второго показателя.

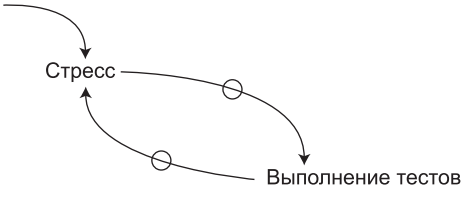

Рис. 25.1. Зловещая спираль «нет времени для тестирования»

Что происходит, когда уровень стресса возрастает?

Чем больший стресс вы ощущаете, тем меньше вы тестируете разрабатываемый код. Чем меньше вы тестируете разрабатываемый код, тем больше ошибок вы допускаете. Чем больше ошибок вы допускаете, тем выше уровень стресса, который вы ощущаете. Получается замкнутый круг с положительной обратной связью: рост стресса приводит к росту стресса.

Что надо сделать, чтобы разорвать этот зловещий цикл? Необходимо либо добавить новый элемент, либо заменить один из элементов, либо изменить стрелки. Попробуем заменить «тестирование» на «автоматическое тестирование».

«Я только что внес в код изменение. Нарушил ли я тем самым его работоспособность?» Рисунок 25.1 показывает динамику в действии. При использовании автоматического тестирования, когда я начинаю ощущать стресс, я запускаю тесты. Тесты превращают страх в скуку. «Нет, я ничего не сломал. Тесты по-прежнему показывают зеленую полосу.» Чем больший стресс я ощущаю, тем чаще я запускаю тесты. Выполнив тесты, я успокаиваюсь. Когда я спокоен, я допускаю меньше ошибок, а это ведет к снижению уровня стресса.

«Да поймите же вы, что у нас нет времени на тестирование!» — теперь эта жалоба перестает быть актуальной, так как выполнение автоматического тестирования почти не требует времени. Компьютер выполняет тестирование значительно быстрее, чем человек. Если вы не выполняете тестирования, вы опасаетесь за корректность кода. Используя автоматическое тестирование, вы можете выбирать удобный для вас уровень страха.

Должны ли вы запустить тест сразу же после его написания, даже если вы полностью уверены, что он не сработает? Конечно, вы можете этого не делать. Но... Приведу поучительный пример. Некоторое время назад я работал с двумя очень умными молодыми программистами над реализацией транзакций, выполняемых внутри

оперативной памяти (это чрезвычайно мощная технология, поддержка которой должна быть добавлена во все современные языки программирования). Перед нами встал вопрос: как реализовать откат транзакции, если начали выполнение транзакции, затем изменили значение нескольких переменных, а затем нарушили ее выполнение (транзакция была уничтожена сборщиком мусора)? Достаточно просто, чтобы проверить способности малоопытных разработчиков. Отойдите в сторону и смотрите, как работает мастер. Вот тест. Теперь подумаем над тем, как заставить его работать. Мы приступили к написанию кода.

Прошло два часа. Два часа, заполненных мучениями и разочарованиями (в большинстве случаев при возникновении ошибки среда разработки давала фатальный сбой и ее приходилось перезапускать). Испробовав множество методов решения проблемы, мы отменили все изменения в коде, восстановили изначальное состояние системы и вернулись к тому, с чего начали: заново написали тот самый тест. На удачу запустили его. Он успешно выполнился. Это было потрясение... Оказалось, что механизм поддержки транзакций на самом деле не менял значений переменных, пока транзакция не считалась полностью выполненной. Надеюсь, теперь вы сами решите для себя, нужно ли вам запускать тесты сразу же после их написания.

Изолированный тест (Isolated Test)

Каким образом выполнение одного теста может повлиять на выполнение другого? Никаким.

Я впервые столкнулся с автоматическим тестированием, когда был еще молодым программистом. В то время в компании с другими программистами (привет, Джоси, привет, Джон!) я занимался разработкой отладчика с графическим интерфейсом. Для контроля корректности его работы использовалась длинная серия автоматических тестов. Это был набор автоматически выполняемых тестов, основанных на взаимодействии с графическим интерфейсом (специальная программа перехватывала нажатия клавиш и события мыши, а затем автоматически воспроизводила их, имитируя работу пользователя с программой). Для выполнения всей серии тестов требовалось длительное время, поэтому обычно тесты запускались вечером, перед уходом с работы, и выполнялись в течение почти всей ночи. Каждое утро, когда я приходил на работу, я видел на своем стуле аккуратно сложенную пачку листов, на которых были распечатаны результаты ночного тестирования. (Привет, Эл!) В удачные дни это мог быть всего один лист, на котором было написано, что ничего не поломалось. В плохие дни на стуле могла лежать огромная кипа бумаги — по одному листу на каждый «сломанный» тест. Постепенно я стал пугаться вида листов бумаги на моем стуле, — если я приходил на работу и видел на своем стуле кипу бумажных листов, меня немедленно бросало в дрожь.

Работая в таком стиле, я пришел к двум важным выводам. Во-первых, тесты должны выполняться достаточно быстро, чтобы я мог запускать их самостоятельно и делать

это достаточно часто. В этом случае я мог бы обнаруживать ошибки раньше, чем кто-либо другой. Во-вторых, спустя некоторое время я заметил, что огромная кипа бумаги далеко не всегда означает огромную кучу проблем. Чаще оказывалось, что в самом начале выполнения тестов один из них завершался неудачей, оставляя систему в непредсказуемом состоянии, из-за чего следующий тест тоже завершался неудачей, а за ним и многие другие — по цепочке.

В то время мы пытались решить эту проблему, автоматически перезапуская систему перед выполнением каждого теста, однако для этого требовалось слишком большое время. Именно тогда у меня возникла еще одна хорошая мысль: тестирование можно выполнять на более низком уровне: вовсе не обязательно, чтобы каждый из тестов выполнялся в отношении всего приложения в целом. Чтобы убедиться в работоспособности всего приложения, достаточно протестировать каждую из его составных частей. Тестирование части приложения можно выполнить быстрее, чем тестирование всего приложения. Однако самый важный вывод состоял в том, что выполнение одного теста никоим образом не должно влиять на выполнение другого теста. Тесты должны полностью игнорировать друг друга. Если один из тестов не срабатывает, это значит, что в программе присутствует одна проблема. Если не срабатывают два теста, значит, в программе присутствуют две проблемы.

Если тесты изолированы друг от друга, значит, порядок их выполнения не имеет значения. Если я хочу выполнить не все, а некоторое подмножество тестов, я не должен беспокоиться о том, что некоторый тест не сработает только потому, что некоторый другой тест не был предварительно запущен.

Производительность является основной причиной, по которой предлагается делать данные общими для нескольких тестов. Требование изоляции тестов принуждает вас разделить проблему на несколько ортогональных измерений, благодаря чему формирование среды для каждого из тестов выполняется достаточно просто и быстро. Иногда, чтобы выполнить подобное разделение, приходится прикладывать значительные усилия. Если вы хотите, чтобы разрабатываемое вами приложение можно было протестировать при помощи набора изолированных друг от друга тестов, вы должны «собрать» это приложение из множества относительно небольших взаимодействующих между собой объектов. Я всегда знал, что это неплохая идея, и всегда радовался, когда мне удавалось реализовать ее на деле, однако я не был знаком ни с одной методикой, которая позволяла бы мне регулярно воплощать эту идею в жизнь. Ситуация изменилась в лучшую сторону после того, как я стал писать изолированные тесты.

Список тестов (Test List)

Что необходимо тестировать? Прежде чем начать, запишите на листке бумаги список всех тестов, которые вам потребуются. Чтобы успешно справляться со стрессом, вы должны постоянно соблюдать важное правило: никогда не делайте

шага вперед, пока не узнаете, в каком месте ваша нога должна коснуться земли. Приступая к сеансу программирования, определите, какие задачи вы намерены решить в ходе этого сеанса.

В рамках весьма распространенной стратегии предлагается держать все в голове. Я пробовал использовать этот подход в течение нескольких лет, однако постоянно сталкивался с одной и той же проблемой. По мере того как я работаю, передо мной возникают все новые и новые задачи, которые необходимо решить. Чем больше задач предстоит решить, тем меньше внимания я уделяю тому, над чем я работаю. Чем меньше внимания я уделяю тому, над чем я работаю, тем меньше задач мне удается решить. Чем меньше задач мне удается решить, тем больше вещей, о которых мне приходится помнить в процессе работы. Замкнутый круг.

Я пытался игнорировать случайные элементы списка и программировать по прихоти, однако это не позволяет разорвать замкнутый круг.

Я выработал привычку записывать на листок бумаги все задачи, которые планирую решить в течение нескольких следующих часов. Этот листок постоянно лежит рядом с моим компьютером. Похожий список задач, которые я планирую решить в течение ближайшей недели или ближайшего месяца, приколот к стене над моим компьютером. Если я записал все эти задачи на бумагу, я уверен в том, что я ничего не забуду. Если передо мной возникает новая задача, я быстро и осознанно решаю, к какому списку («сейчас» или «позднее») она принадлежит и нужно ли вообще ею заниматься.

В контексте разработки через тестирование, список задач — это список тестов, которые мы планируем реализовать. Прежде всего включите в список примеры всех операций, которые требуется реализовать. Далее, для каждой из операций, которые еще не существуют, внесите в список нуль-версию этой операции. Наконец, перечислите в списке все изменения, которые потребуется выполнить, чтобы в конце сеанса программирования получить чистый код.

Но зачем записывать тесты на бумагу, когда можно записать их один за другим в виде готового тестирующего кода? Существует пара причин, по которым я не рекомендую заниматься массовым созданием тестов. Во-первых, каждый из тестов создает некоторую инерцию, мешающую выполнению рефакторинга. Чем больше тестов, тем больше эта инерция. Согласитесь, что выполнить рефакторинг кода, для тестирования которого написаны два теста, сложнее, чем выполнить рефакторинг кода, для тестирования которого написан всего один тест. Конечно, существуют инструменты автоматизированного рефакторинга, которые упрощают эту задачу (например, специальный пункт в меню осуществляет модификацию имени переменной в строке, где она объявляется, и во всех местах, в которых эта переменная используется). Однако представьте, что вы написали десять тестов для некоторого метода и *после* этого обнаружили, что порядок аргументов метода следует изменить на обратный. В подобной ситуации придется приложить существенные усилия, чтобы заставить себя сделать рефакторинг. Во-вторых, чем больше не работающих тестов, тем дольше путь к зеленой полосе. Если перед

вами десять «сломанных» тестов, зеленую полосу вы увидите еще не скоро. Если вы хотите быстро получить перед собой зеленую полосу, вы должны выкинуть все десять тестов. Если же вы хотите добиться успешного выполнения всех этих тестов, вы будете вынуждены долгое время смотреть на красную полосу. Если вы настолько приучены к опрятности и аккуратности кодирования, что не можете позволить себе даже дойти до туалета, пока висит красная полоса, значит, вам предстоит серьезное испытание.

Консервативные скалолазы придерживаются одного важного правила. У человека есть две руки и две ноги, всего четыре конечности, которыми он может цепляться за скалу. В любой момент по крайней мере три конечности должны быть надежно сцеплены со скалой. Динамические перемещения, когда скалолаз перемещает с места на место одновременно две конечности, считаются чрезвычайно опасными. Методика TDD в чистом виде подразумевает использование похожего принципа: в любой момент времени вы должны быть не дальше одного изменения от зеленой полосы.

По мере того как вы заставляете тесты срабатывать, перед вами будет возникать необходимость реализации новых тестов. Заносите эти новые тесты в список задач. То же самое относится и к рефакторингу.

«Это выглядит ужасно <вздох>. Добавим это в список. Мы вернемся к этому перед тем, как завершить работу над задачей.»

Необходимо позаботиться о пунктах, оставшихся в списке на момент завершения сеанса программирования. Если на самом деле вы находитесь в середине процесса реализации некоторой функциональности, воспользуйтесь этим же списком позднее. Если вы обнаружили необходимость выполнения более крупномасштабного рефакторинга, выполнить который в настоящий момент не представляется возможным, внесите его в список «позднее». Я не могу припомнить ситуации, когда мне приходилось переносить реализацию теста в список «позднее». Если я могу придумать тест, который может не сработать, реализация этого теста важнее, чем выпуск кода, над которым я работаю.

Сначала тест (Test First)

Когда нужно писать тесты? Перед тем как вы приступите к написанию тестируемого кода.

Вы не должны выполнять тестирование после. Конечно, вашей основной целью является работающая функциональность. Однако вам необходима методика формирования дизайна, вам нужен метод контроля над объемом работ.

Рассмотрим обычную диаграмму взаимовлияния между стрессом и тестированием (не путать со стресс-тестированием — это совершенно другая вещь): верхний узел — это стресс; он соединяется с тестированием (нижний узел) отрицательной связью;

тестирование, в свою очередь, соединяется со стрессом также отрицательной связью. Эта диаграмма представлена в первом разделе данной главы. Чем больший стресс вы испытываете, тем меньше вы выполняете тестирование. Когда вы знаете, что выполняемого тестирования недостаточно, у вас повышается уровень стресса. Замкнутый цикл с положительной обратной связью. Что можно сделать, чтобы разорвать его?

Что, если мы всегда будем выполнять тестирование вначале? В этом случае мы можем инвертировать диаграмму: вверху будет располагаться узел «Предварительное тестирование», который посредством отрицательной связи будет соединяться с расположенным внизу узлом «Стресс», который, в свою очередь, также посредством отрицательной связи будет соединяться с узлом «Предварительное тестирование».

Когда мы начинаем работу с написания тестов, мы снижаем стресс, а значит, тестирование может быть выполнено более тщательно. Конечно, уровень стресса зависит от множества других факторов, стало быть можно допустить, что возникнет ситуация, в которой из-за высокого уровня стресса нам все-таки придется отказаться от тестирования. Однако, помимо всего прочего, предварительное тестирование является мощным инструментом формирования дизайна и средством контроля над объемом работы. Значит, скорее всего, мы будем выполнять тестирование даже при среднем уровне стресса.

Сначала оператор assert (Assert First)

Когда следует писать оператор `assert`[1]? Попробуйте писать их в первую очередь. Неужели вам не нравится самоподобие?

- С чего следует начать построение системы? С формулировки пожеланий[2] о том, как должна работать система, полученная в результате вашей работы.
- С чего следует начать разработку некоторой функциональности? С написания тестов, которые должны выполниться успешно, когда код будет полностью завершен.
- С чего начать написание теста? С операторов `assert`, которые должны выполняться в ходе тестирования.

С этой методикой познакомил меня Джим Ньюкирк. Когда я начинаю разработку теста с операторов `assert`, я ощущаю мощный упрощающий эффект. Когда вы пишете тест, вы решаете несколько проблем одновременно, даже несмотря на то, что при этом вам не нужно думать о реализации.

---

[1] В переводе с английского языка *assert* означает утверждать, предполагать. Иначе говоря, оператор `assert` фиксирует предположение о прогнозируемом результате теста. — *Примеч. науч. ред.*

[2] Имеются в виду *пожелания пользователей* (user stories). Другой вариант перевода: *пользовательские истории*. — *Примеч. пер.*

- Частью чего является новая функциональность? Является ли она модификацией существующего метода? Является ли она новым методом существующего класса? Является ли она методом с известным именем, но реализованным в другом месте? А может быть, новая функциональность — это новый класс?
- Какие имена присвоить используемым элементам?
- Как можно проверить правильность результата работы кода?
- Что считать правильным результатом работы кода?
- Какие другие тесты можно придумать исходя из данного теста?

Малюсенький мозг, такой как у меня, не сможет хорошо поработать над решением всех этих проблем, если они будут решаться одновременно. Две проблемы из приведенного списка можно легко отделить от всех остальных: «Что считать правильным результатом?» и «Как можно проверить правильность результата?»

Например, представьте, что нам надо реализовать обмен данными с другой системой через сокет. После завершения операции сокет должен быть закрыт, а в буфер должна быть прочитана строка `abc`:

```
testCompleteTransaction() {
   ...
   assertTrue(reader.isClosed());
   assertEquals("abc", reply.contents());
}
```

Откуда должен быть прочитан объект `reply`? Конечно же, из сокета:

```
testCompleteTransaction() {
   ...
   Buffer reply = reader.contents();
   assertTrue(reader.isClosed());
   assertEquals("abc", reply.contents());
}
```

А откуда берется сокет? Мы создаем его, подключаясь к серверу:

```
testCompleteTransaction() {
   ...
   Socket reader = Socket("localhost", defaultPort());
   Buffer reply = reader.contents();
   assertTrue(reader.isClosed());
   assertEquals("abc", reply.contents());
}
```

Однако перед этим мы должны установить соединение с сервером:

```
testCompleteTransaction() {
   Server writer = Server(defaultPort(), "abc");
   Socket reader = Socket("localhost", defaultPort());
   Buffer reply = reader.contents();
```

```
    assertTrue(reader.isClosed());
    assertEquals("abc", reply.contents());
}
```

Теперь мы можем изменить имена в соответствии с используемым контекстом, однако в данном случае мы малюсенькими шажками сформировали набросок теста, генерируя каждое решение в течение пары секунд. Мы начали с написания оператора `assert`.

Тестовые данные (Test Data)

Какие данные следует использовать для предварительных тестов? Используйте данные, которые делают тест простым для чтения и понимания. Помните, что вы пишете тесты для людей. Не разбрасывайте данные в изобилии по всему тесту только потому, что вам хочется добавить в тест как можно больше разнообразных данных. Если в разных местах теста используются разные данные, разница должна быть осмысленной. Если не существует концептуальной разницы между 1 и 2, используйте 1.

Вместе с тем, если ваша система должна поддерживать несколько разновидностей ввода, значит, все эти разновидности должны быть отражены в тестах. Однако не следует использовать в качестве входных данных список из десяти элементов, если при использовании списка из трех элементов будет получен точно такой же дизайн и реализация.

Старайтесь не использовать одну и ту же константу в нескольких местах для обозначения более чем одного понятия. Например, если вы намерены тестировать операцию `plus()`, вам наверняка захочется в качестве теста использовать операцию 2 + 2 — ведь это классический пример сложения. Возможно, вам захочется использовать другую операцию: 1 + 1 — ведь она самая простая из всех возможных. Однако не забывайте, что в данном случае речь идет о двух разных слагаемых, которые могут быть разными объектами. При использовании выражения 2 + 2 слагаемые оказываются одинаковыми, а значит, тест не является достаточно общим. Представьте, что в ходе дальнейшей разработки вы пришли к выводу, что результат выполнения операции `plus()` по тем или иным причинам должен зависеть от порядка слагаемых (сложно представить себе ситуацию, в которой результат сложения зависит от порядка слагаемых, однако может случиться, что операция `plus()` может перестать быть просто сложением, таким образом, общая идея должна быть вам понятной). Чтобы обеспечить более полноценное тестирование, попробуйте использовать 2 в качестве первого аргумента и 3 в качестве второго аргумента (в свое время тест 3 + 4 был классическим начальным тестом при запуске новой виртуальной машины Smalltalk).

Альтернативой шаблону «Тестовые данные» (`Test Data`) является шаблон «Реалистичные данные» (`Realistic Data`), в рамках которого для тестирования

используются данные из реального мира. Реалистичные данные удобно применять в следующих ситуациях:

- вы занимаетесь тестированием системы реального времени, используя цепочки внешних событий, которые возникают в реальных условиях эксплуатации;
- вы сравниваете вывод текущей системы с выводом предыдущей системы (параллельное тестирование);
- вы выполняете рефакторинг кода, имитирующего некоторый реальный процесс, и ожидаете, что после рефакторинга результирующие данные будут в точности такими же, как до рефакторинга, в особенности если речь идет о точности операций с плавающей точкой.

Понятные данные (Evident Data)

Каким образом в тесте можно отразить назначение тех или иных данных? Добавьте в тест ожидаемый и реально полученный результат и попытайтесь сделать отношение между ними понятным. Вы пишете тесты не только для компьютера, но и для читателя. Через несколько дней, месяцев или лет кто-нибудь будет смотреть на ваш код и спрашивать себя: «Что имел в виду этот шутник, когда писал этот запутанный код?» Попробуйте оставить своему читателю как можно больше подсказок, имейте в виду, что этим разочарованным читателем можете оказаться вы сами.

Вот пример. Если мы конвертируем одну валюту в другую, мы берем комиссию 1,5 за выполнение операции. Представьте, что мы обмениваем американские доллары (USD) на британские фунты стерлингов (GBP). Пусть курс обмена будет составлять 2:1. Если мы хотим обменять $100, в результате мы должны получить 50 GBP – 1,5 % = 49,25 GBP. Мы могли бы написать следующий тест:

```
Bank bank = new Bank().
bank.addRate("USD", "GBP", STANDARD_RATE);
bank.commission(STANDARD_COMMISSION);
Money result = bank.convert(new Note(100, "USD"), "GBP");
assertEquals(new Note(49.25, "GBP"), result);
```

Однако вместо этого мы можем сделать порядок вычислений более очевидным:

```
Bank bank = new Bank();
bank.addRate("USD", "GBP", 2);
bank.commission(0.015);
Money result = bank.convert(new Note(100, "USD"), "GBP");
assertEquals(new Note(100 / 2 * (1 - 0.015), "GBP"), result);
```

Прочитав этот тест, я вижу взаимосвязь между входными значениями и значениями, используемыми в составе формулы.

Шаблон «Понятные данные» (`Evident Data`) обладает побочным эффектом: он в некоторой степени облегчает программирование. После того как мы в понятной форме записали выражение `assert`, мы получаем представление о том, что именно нам необходимо запрограммировать. В данном случае мы видим, что тестируемый код должен содержать операции деления и умножения. Мы даже можем воспользоваться шаблоном «Поддельная реализация» (`Fake It`), чтобы узнать, где должна располагаться та или иная операция.

Шаблон «Понятные данные» (`Evident Data`) выглядит как исключение из правила о том, что в коде не должно быть «магических» чисел. Дело в том, что в рамках одного метода легко понять назначение того или иного числа. Однако если в программе уже имеются объявленные символьные константы, я предпочитаю использовать их вместо конкретных численных значений.