Глава 2

Запахи тестов

О чем идет речь в этой главе

В главе 1, “Краткий обзор”, очень сжато описывались основные шаблоны и запахи, рассматриваемые в данной книге. В этой главе более подробно рассматриваются “запахи тестов”, которые встречаются в реальных проектах. Сначала речь идет о базовой концепции запаха теста, а затем — о запахах из трех основных категорий: запахи кода тестов, запахи поведения автоматизированных тестов и запахи проекта, связанные с автоматизированным тестированием.

Введение в запахи тестов

В своей книге *Рефакторинг: улучшение существующего кода* Мартин Фаулер описал несколько способов модификации кода без изменения реализованной функциональности. Причиной такого рефакторинга был назван “неприятный запах”, часто возникающий в объектно-ориентированном коде.

Запахи кода были описаны в главе 3, “Код с душком”, соавтором в которой выступил Кент Бек. Глава начиналась знаменитой цитатой бабушки Бека: “Если воняет, меняй!” Причиной появления этой цитаты был вопрос: “Как узнать, что пришло время менять пеленки?” В результате в лексикон разработчиков был добавлен новый термин.

Запахи кода, описанные в книге Фаулера, характерны для проблем в коде готовых программных продуктов. Многие давно подозревали, что существуют запахи, встречающиеся только в коде автоматизированных тестов. Публикация доклада *Рефакторинг тестового кода* [RTC] на конференции XP2001 подтвердила эти подозрения. В докладе был идентифицирован ряд “неприятных запахов”, возникающих только в тестовом коде. Авторы доклада предложили ряд рефакторингов, которые позволяют избавиться от ядовитых запахов.

В этой главе приводится обзор *запахов тестов* (test smells). Более подробные примеры каждого запаха приводятся в соответствующих главах.

Что такое запах теста

Запах является симптомом проблемы. Запах не обязательно указывает на конкретную ошибку, так как может исходить из нескольких источников. Большинство запахов в дан-

ной книге имеют несколько причин. Некоторые причины приводят к появлению нескольких запахов. Это связано с тем, что основная причина может проявлятся в нескольких симптомах (т.е. запахах).

Не все проблемы считаются запахами, а некоторые проблемы могут даже быть причиной запахов. Тест "бритва Оккама" позволяет определить, запах это или проблема (т.е. запах должен заставить разработчика схватиться за нос, показывая, что "здесь что-то не так"). Как будет продемонстрировано в следующем разделе, запахи классифицированы по проявляемым симптомам (по способу "хватания за нос").

На основании этого критерия некоторые запахи тестов из предыдущих публикаций и статей были понижены до уровня "причин". Их названия в большинстве случаев не изменились, поэтому они упоминаются в описании побочных эффектов применения шаблонов. В таком случае проще рассматривать непосредственно причину, а не более общий, но заметный запах.

Типы запахов тестов

С течением времени было обнаружено, что существует как минимум два типа запахов: **запахи кода** (code smells), которые распознаются в ходе анализа кода, и **запахи поведения** (behavior smell), которые влияют на результат выполненного теста.

Запахи кода представляют собой **антишаблоны** на уровне создания кода. Разработчик, тестер или преподаватель может заметить их при чтении или написании кода тестов, т.е. код просто выглядит не совсем правильно или не доносит намерение достаточно ясно. Запахи кода сначала необходимо распознать и только потом действовать. При этом необходимость такого действия не всем может показаться очевидной. Запахи кода относятся ко всем тестам, включая *тесты на основе сценариев* (Scripted Test, с. 319) и *записанные тесты* (Recorded Test, с. 312). Особенно они характерны для записанных тестов, в которых приходится сопровождать записанный код. К сожалению, большинство записанных тестов превращается в *непонятные тесты* (Obscure Test, с. 230), так как они записаны утилитой, не рассчитанной на генерацию читаемого людьми кода.

С другой стороны, запахи поведения намного сложнее игнорировать, так как из-за них тесты завершаются неудачно (или вообще не компилируются) в самый неожиданный момент, например при попытке интеграции кода в важную версию; в таком случае придется обнаружить причину проблемы, прежде чем ее решать. Как и запахи кода, запахи поведения касаются тестов на основе сценариев и записанных тестов.

Обычно разработчики замечают запахи кода и поведения во время автоматизации, сопровождения и запуска тестов. В последнее время был обнаружен еще один тип запаха — обычно этот запах выявляет руководитель проекта или потребитель, который не имеет доступа к коду тестов и не запускает тесты. **Запахи проекта** (project smell) являются индикаторами общего состояния проекта.

Что делать с запахами

Появления некоторых запахов не избежать, так как они требуют слишком больших трудозатрат на устранение. Важно осознавать присутствие запахов и понимать причины их возникновения. После этого можно принять осознанное решение о выборе запахов, которые будут устранены для нормального завершения проекта.

Решение о списке удаляемых запахов принимается на основе баланса между стоимостью трудозатрат и полученной выгодой. От одних запахов избавиться сложнее; другие вызывают больше проблем. Необходимо избавиться от запахов, которые имеют наибольший отрицательный эффект, так как они не позволят нормально завершить работу. Как уже было сказано, многих запахов можно избежать, если выбрать подходящую стратегию автоматизации и следовать хорошим стандартам оформления кода автоматизированных тестов.

Хотя здесь тщательно описываются различные типы запахов, важно обратить внимание, что часто одновременно наблюдаются симптомы запахов каждого типа. Например, запахи проекта представляют собой симптом проблемы на более низком уровне. Проблема может проявляться как запах поведения, но реальной причиной проблемы, скорее всего, является еще более низкоуровневый запах кода. Хорошая новость: существуют три различных способа идентификации проблемы. Плохая новость: очень легко сконцентрироваться на симптоме более высокого уровня и попытаться решить проблему, не разобравшись в реальной причине происходящего.

Эффективной методикой определения причины является *пять "почему"*. Сначала необходимо узнать, почему что-то происходит. После идентификации факторов, ставших причиной происходящего, нужно определить, почему возник каждый из этих факторов. Этот процесс повторяется до тех пор, пока не будет получена новая информация. На практике пяти итераций "почему" оказывается достаточно — отсюда и название методики *пять "почему"*[1].

В оставшейся части этой главы рассматриваются запахи, которые с большей вероятностью возникают при работе над проектами. Начнем с запахов проектов. После этого будут рассмотрены запахи поведения и запахи кода, которые являются их причиной.

Каталог запахов

Разобравшись, что такое запахи и какую роль они играют в проектах, использующих автоматизированное тестирование, рассмотрим примеры таких запахов. Описание отдельных запахов и причин их возникновения приводятся в части II.

Запахи проектов

Запахи проектов являются симптомами ошибок в самом проекте. Причиной таких запахов может быть запах поведения или кода. Но так как руководители проектов редко пишут или запускают тесты, запахи проектов будут для них первым сигналом о снижении качества части проекта, связанной с автоматизацией тестов.

Основное внимание руководители проектов уделяют функциональности, качеству, ресурсам и стоимости. Именно по этой причине запахи проекта, скорее всего, будут касаться этих характеристик. Наиболее очевидной для руководителя проекта метрикой, которая будет воспринята, как запах, является качество программного обеспечения, которое измеряется в виде количества дефектов, обнаруженных во время формального тестирования или при использовании потребителем. Если количество *ошибок в продукте* (Production Bugs, с. 303) оказывается бо́льшим, чем ожидалось, руководи-

[1] Эта методика также называется анализом причин или "очисткой лука".

тель проекта должен задать вопрос: “Почему эти ошибки прошли через фильтр автоматизированных тестов?”

Руководитель проекта может следить за количеством неудачных ежедневных интеграционных компиляций, чтобы получить своевременную индикацию качества программного обеспечения и следования выбранному групповому процессу разработки. Руководитель должен начинать беспокоиться, если интеграция завершается неудачей слишком часто (особенно если проблема не решается за несколько минут). **Анализ причин** (root cause analysis) возникновения ошибок может показать, что многие проблемы не связаны с ошибками в самом программном обеспечении, а являются следствием *тестов с ошибками* (Buggy Test, с. 296). Это пример, когда тесты кричат “Караул!” и требуют ресурсов для исправления ситуации, но не повышают качества **кода продукта** (production code).

Тест с ошибками (Buggy Test) является только одним из компонентов более общей проблемы *высокой стоимости обслуживания тестов* (High Test Maintenance Cost, с. 300), что может заметно повлиять на производительность команды разработчиков. Если тесты должны модифицироваться слишком часто (т.е. при каждой модификации тестируемой системы) или если стоимость модификации слишком высока из-за непонятных тестов, руководитель проекта может принять решение о перенаправлении ресурсов с написания автоматизированных тестов на создание большего объема кода продукта или тестирование вручную. Скорее всего, на этом этапе руководитель проекта запретит разработчикам писать тесты. (Достаточно сложно заставить руководителя проекта разрешить разработчикам писать автоматизированные тесты. Но если это все-таки удалось сделать, не стоит терять данную возможность из-за небрежности или неэффективности. Скорее всего, именно по этой причине и была написана данная книга: помочь разработчикам убедить руководителя проекта и не дать им повода отказаться от автоматизированного модульного тестирования.)

С другой стороны, руководитель проекта может решить, что *ошибки в продукте* (Production Bugs) стали следствием того, что *разработчики не пишут тесты* (Developers Not Writing Tests, с. 298). Такое заявление обычно делается во время ретроспективного анализа или анализа причин проблемы. Отказ от написания тестов может быть вызван слишком жестким графиком разработки, непосредственными руководителями, которые рекомендуют “не тратить время на тесты”, или разработчиками, не обладающими достаточной квалификацией для написания тестов. Среди других возможных причин можно назвать навязанный дизайн, плохо поддающийся тестированию, или среду тестирования, которая способствует появлению *“хрупких” тестов* (Fragile Test, с. 277). Наконец, эта проблема может быть следствием *потерянных тестов* (Lost Test) — необходимые тесты существуют, но не включены в *набор всех тестов* (AllTests Suite; см. *Именованный набор тестов*, Named Test Suite, с. 604), который используется разработчиками перед включением изменений в общее хранилище кода или вызывается инструментами автоматизированной компиляции.

Запахи поведения

Запахи поведения проявляются при компиляции и запуске тестов. Чтобы их заметить, сверхвнимательность не требуется, так как они проявляются в виде ошибок компиляции и неудачного завершения тестов.

Самым распространенным запахом поведения является *“хрупкий” тест* (Fragile Test). Его проявлением является тест, по какой-то причине завершающийся неудачно, но до

этого работавший нормально. Проблема "хрупких" тестов стала причиной неприятия автоматизированных тестов в некоторых кругах. Особенно это касается коммерческих инструментов тестирования, которые работают по принципу "записи и воспроизведения" и обещают простую автоматизацию тестов. Сразу после записи такие тесты очень уязвимы. Часто единственным способом решения проблемы является повторная запись теста, так как записанный код очень сложно читать и модифицировать вручную.

Основные причины возникновения этого запаха можно разделить на четыре широкие категории.

- *Чувствительность к интерфейсу* (Interface Sensitivity) возникает, когда работа теста нарушается из-за изменений в программном или пользовательском интерфейсе, который использовался для автоматизации теста. Коммерческие инструменты "записи и воспроизведения" обычно взаимодействуют с системой через пользовательский интерфейс. Минимальные модификации интерфейса могут привести к неудачному завершению тестов даже в ситуации, в которой пользователь-человек счел бы тест пройденным.
- *Чувствительность к поведению* (Behavior Sensitivity) возникает, когда тест завершается неудачно из-за изменений в поведении тестируемой системы. Конечно, если система меняется, тесты должны завершаться неудачно. Но проблема в том, что на небольшое изменение должно реагировать небольшое количество тестов. Проблема возникает, когда в ответ на небольшое изменение неудачно завершается большинство тестов или вообще все тесты.
- *Чувствительность к данным* (Data Sensitivity) возникает, когда тесты завершаются неудачно после изменения данных, уже присутствующих в тестируемой системе. Такая проблема особенно характерна для приложений, использующих базы данных. Чувствительность к данным является специальным случаем чувствительности к контексту, когда в качестве контекста выступает база данных.
- *Чувствительность к контексту* (Context Sensitivity) возникает, когда тесты завершаются неудачно из-за изменений в окружении тестируемой системы. Наиболее частым примером является зависимость тестов от даты и времени, но эта проблема может возникать и в результате зависимости от состояния устройств (серверов, принтеров или мониторов).

Чувствительность к данным и чувствительность к контексту являются примерами специального типа *"хрупкого" теста* (Fragile Test), известного как *"хрупкая" тестовая конфигурация* (Fragile Fixture). При этом модификация часто используемой тестовой конфигурации приводит к неудачному завершению нескольких существующих тестов. Такой сценарий увеличивает стоимость расширения *стандартной тестовой конфигурации* (Standard Fixture, с. 338) для поддержки новых тестов, а значит, делает невыгодным создание тестов для всех компонентов продукта. Хотя основной причиной "хрупкости" конфигурации является плохой дизайн тестов, проблема проявляется в результате изменения тестовой конфигурации, а не после изменений в тестируемой системе.

В большинстве проектов с гибкой разработкой применяется ежедневная или **непрерывная интеграция** (continuous integration), состоящая из двух этапов: компиляции последней версии исходного кода и запуска всех автоматизированных тестов над скомпилированным продуктом. *Рулетка утверждений* (Assertion Roulette, с. 264) может затруднить определение, как и почему тесты завершаются неудачно во время интеграции и

какие утверждения оказались ошибочными. Это связано с недостаточностью информации в журнале ошибок. Диагностика ошибок во время компиляции может потребовать значительного времени, так как ошибку необходимо воспроизвести в среде разработки и только после этого выполнять поиск причины ошибки.

Распространенным источником неприятностей является неудачное завершение тестов без видимых причин, т.е. ни код тестируемой системы, ни код тестов не модифицировались, а тест неожиданно начинает завершаться неудачно. При попытке воспроизвести результат в среде разработки тест может завершиться успешно или неудачно. Такие *нестабильные тесты* (Erratic Test, с. 267) очень раздражают разработчиков и требуют много времени для исправления, так как причин некорректного поведения может быть множество. Некоторые из них перечислены ниже.

- *Взаимодействующие тесты* (Interacting Tests) возникают, когда несколько тестов используют *общую тестовую конфигурацию* (Shared Fixture, с. 350). В таком случае очень сложно запустить тесты по отдельности или запустить несколько наборов тестов в пределах большего *набора наборов* (Suite of Suites). Также это может привести к каскадным отказам, когда неудачное завершение одного теста приводит *стандартную тестовую конфигурацию* (Standard Fixture) в состояние, заставляющее завершиться неудачно множество других тестов.
- *"Война" запуска тестов* (Test Run War) возникает, когда несколько *программ запуска тестов* (Test Runner, с. 405) одновременно запускают тесты с использованием *общей тестовой конфигурации* (Shared Fixture). Обычно это происходит в самое неподходящее время, например при попытке исправить последние несколько ошибок перед выпуском следующей версии.
- *Неповторяемые тесты* (Unrepeatable Test) возвращают разные результаты при каждом запуске теста. Это может вынудить разработчика прибегнуть к *ручному вмешательству* (Manual Intervention, с. 287).

Частая отладка (Frequent Debugging, с. 285) является еще одним запахом, снижающим производительность труда. Автоматизированные модульные тесты должны избавить от необходимости пользоваться отладчиком в большинстве случаев, так как список неудачно завершившихся тестов явно указывает на источник ошибки. Запах *частая отладка* (Frequent Debugging) указывает, что модульные тесты недостаточно покрывают существующий код или тест проверяет слишком большую часть функциональности.

Реальная ценность *полностью автоматизированных тестов* (Fully Automated Test, с. 81) заключается в максимальной частоте их запуска. При разработке на основе тестов разработчик может запускать весь набор или часть тестов один раз в несколько минут. Такое поведение стоит стимулировать, так как снижается время обратной связи, а значит, и стоимость устранения дефектов кода. Если тесты требуют *ручного вмешательства* (Manual Intervention) при каждом запуске, разработчики стараются запускать тесты реже. Такая практика увеличивает стоимость обнаружения всех дефектов, внесенных с момента последнего запуска теста.

Такое же отрицательное действие на производительность оказывает запах *медленные тесты* (Slow Tests, с. 289). Если запуск тестов требует более 30 секунд, разработчики не будут запускать тесты после внесения каждой модификации. Вместо этого будет выбираться "логичный момент" для запуска, например перед перерывом на кофе, перед обедом или перед совещанием. Отложенная обратная связь приводит к потере "непрерывности" и увели-

чению времени между внесением и обнаружением дефекта. Наиболее распространенное решение для избавления от этого запаха является и наиболее проблемным; применение *общей тестовой конфигурации* (Shared Fixture) может привести к появлению различных запахов поведения и должно рассматриваться как крайняя мера.

Запахи кода

Запахи кода являются "классическими" запахами, описанными в книге Мартина Фаулера *Рефакторинг* [Ref]. Большинство запахов, рассмотренных Фаулером, относились к коду. Они должны распознаваться авторами тестов в процессе обслуживания кода. Хотя запахи кода обычно влияют на стоимость обслуживания тестов, их можно рассматривать как ранние признаки появления запахов поведения в будущем.

При чтении тестов достаточно очевидным (и часто игнорируемым) запахом является *непонятный тест* (Obscure Test). Он может принимать разные формы, но все версии имеют один и тот же эффект: очень сложно определить, что же делает тест, так как тест не *доносит намерение* (Communicate Intent, с. 95). Такая неопределенность увеличивает стоимость обслуживания тестов и может привести к появлению *тестов с ошибками* (Buggy Test), когда **ответственный за тест** (test maintainer) вносит некорректные изменения в код теста.

Еще одним характерным запахом является *условная логика теста* (Conditional Test Logic, с. 243). Тесты должны быть простыми, линейными последовательностями операторов. Если тест имеет несколько ветвей выполнения, нельзя знать точно, по какому пути пойдет выполнение теста в конкретном случае.

Запах *фиксированные данные теста* (Hard-Coded Test Data) может стать очень опасным по ряду причин. Во-первых, тест сложнее понять: приходится рассматривать каждое значение и угадывать, связано ли оно с другими значениями, чтобы спрогнозировать поведение тестируемой системы. Во-вторых, возникает проблема, если тестируемая система включает в себя базу данных. Запах *фиксированные данные теста* (Hard-Coded Test Data) может привести к появлению *нестабильных тестов* (Erratic Test), если тест использует уже существующий в базе данных идентификатор, или *"хрупкой" тестовой конфигурации* (Fragile Fixture), если значения ссылаются на уже модифицированные записи в базе данных.

Запах *сложный в тестировании код* (Hard-to-Test Code, с. 251) может оказаться фактором, способствующим появлению других запахов кода и поведения. Эта проблема наиболее очевидна для автора тестов, который не может найти способ создания тестовой конфигурации, вызова методов тестируемой системы или проверки ожидаемого результата. В итоге автору тестов приходится проверять больший объем кода (большую тестируемую систему с большим количеством классов), чем хотелось бы. При чтении кода теста данный запах проявляется как *непонятный тест* (Obscure Test), поскольку автору теста приходится обходить различные ограничения для взаимодействия с тестируемой системой.

Запах *дублирование тестового кода* (Test Code Duplication, с. 254) является плохой практикой, так как он увеличивает стоимость обслуживания тестов. В результате приходится обслуживать больше кода и код оказывается сложнее, так как зачастую этот запах сопровождается запахом *непонятный тест* (Obscure Test). Такая ситуация возникает, когда автор теста просто клонирует его и не задумывается о более осмысленном повторном использовании логики теста[2]. С ростом потребности в тестировании от автора тестов тре-

[2] Обратите внимание, что написано "повторном использовании логики", а не "повторном использовании методов".

буется выделение часто используемых последовательностей операторов во *вспомогательные методы теста* (Test Utility Method, с. 610), которые можно использовать повторно в различных *тестовых методах* (Test Method, с. 378)[3]. Такой подход снижает стоимость обслуживания тестов.

Запах *логика теста в продукте* (Test Logic in Production, с. 257) является нежелательным, так как нет гарантии невозможности случайного запуска этого кода[4]. Кроме того, код готового продукта становится больше и сложнее. Наконец, такая ошибка может привести к включению в выполняемый файл дополнительных программных компонентов и библиотек.

Что дальше

В этой главе были описаны различные неприятности, происходящие при автоматизации тестов. В главе 3, "Цели автоматизации", рассматриваются цели, о которых необходимо помнить при создании автоматизированных тестов. Понимание целей позволит подготовиться к пониманию принципов. А следование принципам позволит обойти большинство проблем, перечисленных в данной главе.

[3] Очень важно не использовать *тестовые методы* (Test Method) повторно, так как в результате получается *гибкий тест* (Flexible Test; см. *Условная логика тестов*, Conditional Test Logic).
[4] Обратите внимание на врезку о ракете Ariane на с. 257 с поучительной историей.